Object-Oriented Structuring of Finite Elements

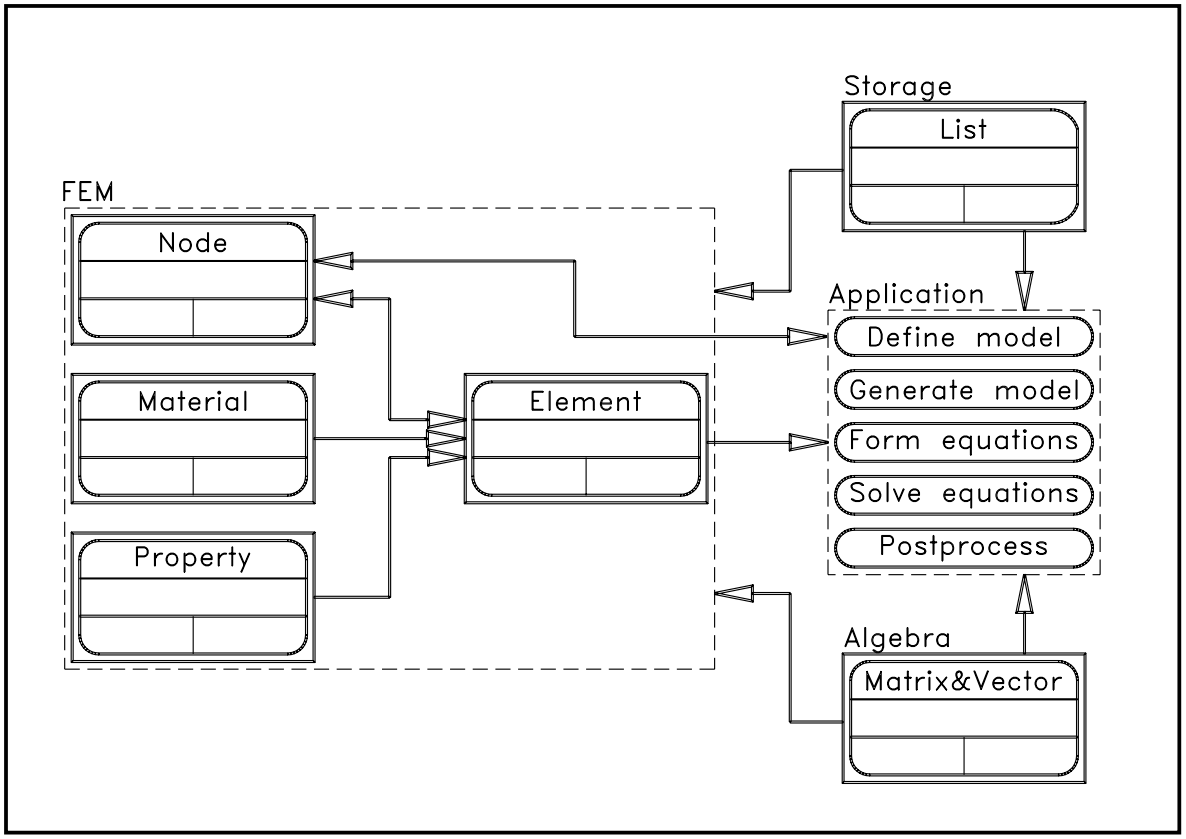
Hededal, O.

Publication date:
1994

Document Version
Publisher's PDF, also known as Version of record

OBJECT-ORIENTED STRUCTURING OF FINITE ELEMENTS

OLE HEDEDAL



AALBORG UNIVERSITY

September 1994

Acknowledgements

This thesis, *Object-oriented Structuring of Finite Elements*, has been prepared in connection with a Ph.D. study carried out in the period September 1991 to April 1994 at the Department of Building Technology and Structural Engineering, University of Aalborg, Denmark.

A visit at the Department of Structural Engineering, Chalmers Technical University, Sweden, in the period March to May 1993 was used for writing a major part of the program code. I thank Professor N.E. Wiberg and my colleagues for making the visit a professional as well as social success.

I greatly acknowledge the inspiration and guidance given to me by my supervisor Professor, Dr. Techn. Steen Krenk. Furthermore, I thank Norma Hornung for assistance in the preparation of the illustrations in the thesis.

The work has been financed by a grant from the Technical Faculty at the University of Aalborg. The visit to Chalmers Technical University was supported by Nordisk Forskerutdanningsakademi (NorFA).

Aalborg, September 1994 Ole Hededal

Contents

Chapter 1

Introduction

1.1 Motivation

The need for highly specialized finite element codes is today present in research, practical engineering and education. While many problems can be solved using standard codes, there is a variety of problems that require special elements or material models or for which the available solution algorithms are not efficient or stable enough. In such situations the user is interested in easy ways to implement new elements or to obtain stable algorithms, i.e. the user wants to provide new facilities with the least possible effort.

In most cases only limited modifications are needed. It could for example be the change of a single parameter in order to monitor its influence on the result or a slight change in the element formulation. Implementation of an entire new element or material model, either for research purposes or for solving problems that are not standard, requires greater changes in the code concerning both computation and data management. The computation of the model parameters is usually well-defined and easy to program and test. Most of the new code will, however, be used for the data management, e.g. input/output and storage of data; this is trivial but nonetheless tedious and prone to errors.

Another essential part of a finite element program is the solution algorithms. The algorithms may be divided in solution methods and solution strategies. The methods usually concern basic matrix manipulations such as solution of linear equation systems or solution of an eigenvalue problem. They are characterized by a close relation to the structure of the system matrices and their internal representation, i.e. different methods should be employed for non-symmetric, symmetric, banded or sparse matrices. The strategies are for example solution of non-linear equations and time integration schemes. They use information generated by the element procedures and the solution methods to seek a solution iteratively. A strategy consists of a number of controls that the user can manipulate or change in order to stabilize the algorithm or make it more efficient. The program should allow the user to choose freely the methods and strategies that most efficiently solve the considered problem.

The issues described above have 3 levels of abstraction. The solution methods may be referred to as low level programming and are mainly dictated by the matrix structure. The medium level concerns implementation of new elements and materials and involves

modification or addition of procedures that describe a part of the physical problem. The strategies have the highest level of abstraction. They are mathematical instruments that are used to obtain a solution of the global system built of elements and materials. Implementation of new strategies consists of modifying the control structures, that uses existing procedures, rather than adding new procedures.

1.1.1 Numerical requirements

From a numerical point of view a finite element code should ideally possess

- Efficiency
- Robustness
- Flexibility

3 groups of finite element codes will be considered discussing these properties in relation to the possibilities of incorporating new facilities in the existing finite element code: commercial general-purpose programs, specialized programs made by the user and systems that use high level languages. The first 2 groups are compiled codes that use algorithmic techniques, while the last type uses high level command driven languages.

Commercial general-purpose finite element programs like ANSYS (1988) and ABAQUS (1992) consist of a precompiled main processor that includes the implementation of standard elements, material models and solution procedures. In order to perform an analysis the user supplies a command file (written as a text file or generated by a graphical preprocessor) that is interpreted by a driver. The driver generates a Fortran code that describes the finite element model in terms of the program variables and calls to the involved procedures. This program segment is then compiled and linked along with the main processor to an executable file. The analysis is performed running the execute file. The user is allowed to supply subroutines that specifies a special solution strategy or implements a user-defined element. The user subroutines are compiled and linked along with the rest of the code. The advantage of using programs like ANSYS and ABAQUS for analysing non-standard problems is that the user can benefit from their strong capabilities e.g. concerning solution of large problems, advanced elements and graphical presentation. Many resources have been invested in optimizing and verifying the implementations of elements and algorithms in these codes to ensure the quality and robustness of the analysis results. Therefore, to protect the commercial interests the source codes are not available to the common user. The drawback of this is that it restricts the possibility of monitoring special variables or it might even restrain the user from trying to make a user subroutine.

Another class of finite element programs are the programs that are developed by the user to treat special problems. These programs are usually modest compared to the general-purpose programs with respect to the number of available elements and solution strategies, the capability of running large problems and the graphical presentation. The advantage of open codes is that the user have access to all parts of the program and can access any needed parameter or modify any procedure in order to satisfy the current needs. These codes are traditionally programmed in procedural languages like Fortran, Pascal or C, which have shown superior for numerical computation. Large procedural programs, however, tend to grow complex due to the organization of the data and procedures. To enhance the

readability the programs can use data structures to collect related data in a single entity, e.g. a node structure which consists of the node number, the node coordinates and the nodal displacements and loads. Further, the programmer may want to use object-oriented programming in which related data and methods are collected in an entity - the object. Hereby the relation between a variable and its function in the problem becomes more clear. Object-oriented programming is therefore a way to improve the program structure of a finite element codes, making modification and extension simpler. For the numerical parts procedural programming could still be used.

The finite element code can also be programmed in a high level language like in Matlab, CALFEM (1993), Dahlblom et al. (1985), or even in a symbolic language like in Maple, Beltzer (1990). In the Matlab implementation of CALFEM the element formulation is given in a toolbox that is used in combination with the existing facilities. The toolbox consists of a set of functions, e.g. generation of the element stiffness matrices or calculation of the strain. The analysis is performed interactively with the user defining the model, generating the system matrices, calling the (built-in) solution algorithm and finally presenting the results in tables or graphically. The user is thus always in control of the process and may stop the analysis at any time to preview parameters. Also the direct access to graphical tools motivate the use of such programs for educational and development work. However, for larger problems efficiency requirements may limit the use of this type of program.

1.1.2 Requirements to the program structure

Developing a finite element program requires strategy and program structuring. Planning a program system three issues should be taken into account,

- Specialization
- Expansion
- Maintenance

In order for more people to work with same program system - simultaneously or over a period of several years - it is important to make an expandable framework where different parts can be developed independently without affecting other parts of the system. Therefore there should be very few bindings between the different parts. This would allow specialization and expansion of the program system with more efficient algorithms, new elements or additional postprocessing facilities. The program system thereby becomes capable of handling a large number of problems without increasing the complexity of the program structure. Furthermore, large-scale programs typically have lifetimes that outlast the immediate involvement of the single programmer. It is therefore important that new programmers can take over the code maintenance and development without having to rely on several years of experience with the system.

In order to maintain and specialize the program code to meet new requirements it is necessary to have an architecture that is flexible and does not restrict modifications. First of all, it is useful to divide the program into modules. This enhances readability and provides an easier overview of the program structure. Second, related variables should be gathered in data structures in order to limit the number of global variables. Along with the expansion of the computer power and storage the program codes have expanded equally.

Therefore, it is important to be able to reuse code as this would save time and limit the possibility of introducing errors. These requirements can be met concentrating on

- Modularity
- Data structuring
- Code reuse

Object-oriented programming supports modularity and data structuring by organizing the problem in objects with very few internal bindings. Objects can inherit functionality from each other, thus using the same code for several types of problems.

1.1.3 Aim

The aim of this thesis is to describe an implementation of an open, expandable framework, ObjectFEM, that permits the user to choose freely among the available set of elements, materials and algorithms or provide new ones. A program structure with three levels is defined using object-oriented structuring and programming. The top level concerns the algorithms used to solve the global equations. i.e. possibilities of solving linear as well as non-linear equation systems. The top level can be referred to as the application and it is at this level the user can define new solution algorithms. Level 2 is used to describe the finite element theory. At this level different elements and materials may be implemented using a standard programming interface and directly used in already existing applications. The lowest level contains the definition of tools. In scientific programming the most important tool is linear algebraic classes such as vectors and matrices. However, others such as lists, arrays or graphics may be useful. Level 3 deals with formulation of such tools.

The structure of this thesis tries to emphasize the step-by-step style in which object-oriented finite element programs can be developed. Having identified the central concepts in a finite element formulation a basic framework is described. The basic framework defines a standard interface for programming new elements or materials. In connection to this an application for solution of linear static problems is presented. The framework must be customized to apply to a specific problem. This is done considering isoparametric continuum elements for potential problems and linear elasticity theory. Solution of non-linear finite element problems requires iterative strategies. Different solution strategies are considered in order to identify extensions to the linear framework and an application for solution of non-linear problems is presented. A geometrically non-linear bar element is formulated as an example of a simple non-linear element. Elasto-plastic material models are used to identify further extensions of the standard framework to deal with path-dependent material models and von Mises associated plasticity will be implemented as an illustration of the concepts.

1.2 The structure of finite elements

A finite element program can be described in terms of

- Data
- Methods

- Algorithms

The data are the variables that contain the model description, e.g. node labels, element topology or material properties, and the variables that store the system matrices and vectors, such as the global stiffness matrix or load vector. Four groups of methods manipulate the data: FEM methods, data management methods, I/O methods and solution methods. The FEM methods relate directly to the finite element formulation and are responsible for calculating the element stiffness matrix or the strain occurring from the applied loads. Data management mainly consists of storing the model definition and the analysis results, but involves also the generation and assembly of the global system, e.g. assembly of the global stiffness matrix. I/O methods take care of input and output from the program. The program must be able to obtain its input from a file or a graphical user interface and present the analysis results in tables or graphics. The solution methods are related to the global matrices and vectors and are used for solving the linear equation systems or obtaining the eigenvalues of the system. Algorithms are in this context the strategies that are used for controlling an analysis such as linear analysis programs or strategies for solving non-linear equation systems.

ALGORITHM 1.1: TRUSS ANALYSIS PROGRAM

```
Variables:
    node_no, elem_no, matl_no
    coor, topo, prop
    a, f, fix, Ke, K, N

Preprocess:
    read_node(node_no,coor)
    read_elem(elem_no,topo)
    read_matl(matl_no,prop)
    read_bc(a,fix,f)

    for (all elements) do
      elem_stiffness(elem_no,topo,coor,prop,Ke)
      assm_stiffness(elem_no,topo,Ke,K)

Process:
    factor(K,fix)
    solve(K,a,f,fix)

Postprocess:
    for (all nodes) do
      write_disp(node_no,a)
      write_load(node_no,f)

    for (all elements) do
      elem_force(elem_no,topo,coor,prop,a,N)
      write_elem_force(elem_no,N)
```

1.2.1 Sample program

A simple finite element program for truss analysis with bar elements given as Algorithm 1.1 can be used to illustrate the program structure. It is an example of an algorithm for linear analysis. An algorithm is built of control structures: `for` (all elements) `do` symbolizes a loop over all elements in the model. Standard structures like `for-do` and `while-do` are written in `courier`. Variables and methods are written in sans serif. Variables that are needed by the methods are given in the argument list which is enclosed in parenthesis: $(\cdot)$.

Classifying the program in data and groups of methods gives

Data:

node_no,elem_no,matl_no: Identification of nodes, elements and materials by a number.

coor: Node coordinates.

topo: Element topology: contains the numbers of the connected nodes and material.

prop: Material properties: elastic modulus and cross section area of the bar.

a: Vector of nodal displacements - the degrees-of-freedom ordered by their position in the global system.

f: Vector of nodal loads - ordered as a.

fix: Array indicating if a degree-of-freedom is prescribed.

Ke: Temporary matrix that stores the element stiffness matrix.

K: Global stiffness matrix.

N: Element section force: axial force.

FEM methods:

elem_stiffness: Calculates the element stiffness matrix and returns it in terms of the global coordinate system.

elem_force: Calculates the axial force in the bar.

Data management methods:

assm_stiffness: Assembles the global stiffness matrix from the element stiffness matrices using the topology information.

I/O methods:

read_node: Reads the node number and coordinates

read_elem: Reads the element number and topology.

read_matl: Reads the material number and the properties, E and A.

read_bc: Reads the boundary conditions: prescribed displacements and loads.

write_disp: Writes the nodal displacements.

write_load: Writes the nodal loads.

write_elem_force: Writes the calculated axial force.

Solution methods:

factor: Factorizes the constrained matrix K

solve: Solves the constrained equation system Ka=f.

Algorithm 1.1 follows the standard structure of a procedural finite element programs. First it identifies the data and then calls a number of methods in reaching the solution. In

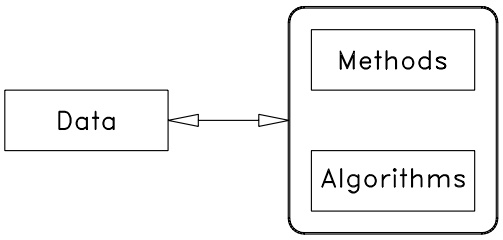


Figure 1.1: Organization in procedural programs

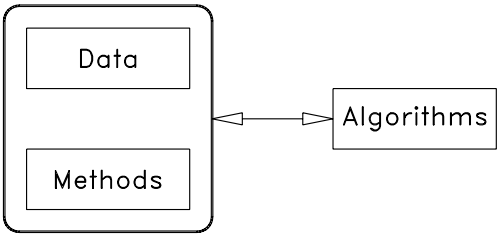


Figure 1.2: Organization in object-oriented programs

procedural programming the data are grouped separately from the methods and algorithms as illustrated by Figure 1.1.

Looking at the data of a finite element program it is immediately recognized that they group naturally in the 3 data structures Node, Element and Material and the data that contains the global model, K, a, f and fix. The methods can be divided by their relation to the data structures, thereby grouping the data and methods together while the algorithms still are independent. An entity that contain data and the related methods is called an object. The organization in object-oriented programs is illustrated in Figure 1.2.

In the finite element method 3 built-in concepts are the obvious object candidates: Node, Element and Material. An object is a self-contained entity, i.e. it must be able to do its own I/O, manage itself and maybe represent parts of the finite element formulation. The Node is mainly responsible for managing degrees-of-freedom and point loads and sending the prescribed values to the global system. The Element must for example be able to compute the element stiffness matrix and the Material must generally be able to represent the constitutive behaviour in a matrix form and present it to the elements. Other candidates for objects are matrices and vectors, for these there must be methods that perform algebraic operations like additions and multiplications, but also methods that can solve a linear equation system should be available. The structure of a simple object-oriented finite element program is given in Figure 1.3.

The conclusion to the discussion above is that a finite element program consists of 4 types of operations that should be taken care of

- Model I/O
- Data management
- Problem formulation - elements and materials
- Solution methods and strategies

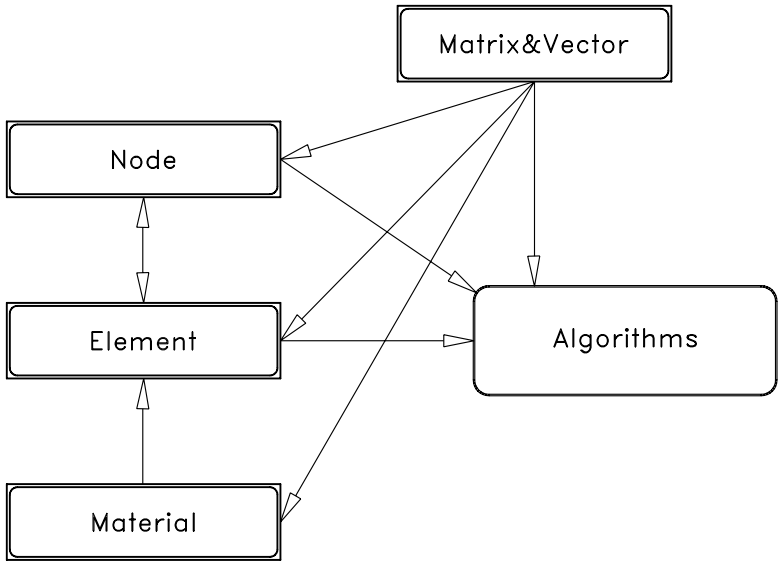


Figure 1.3: Object-oriented finite element programs

The programmer should ideally be able to concentrate on either a single problem formulation, solution method or strategy at the time. Using an object-oriented program structure this can be accomplished by inheriting the other parts of the program, especially the data management and I/O methods, from objects that are already defined.

1.3 Object-oriented programming

Object-oriented analysis is a structuring technique that has recently grown popular due to the appearance of languages that support object-oriented programming. Object-oriented analysis is a general tool used to analyze the nature of a problem to be solved in order to obtain a program structure that is highly distributed with very few bindings between different modules. In this section some basic terms of object-oriented analysis and programming are presented and shortly illustrated with finite element concepts. The definitions specifically relate to the way the terms are used later on. In many situations graphical presentation is the best way to describe the architecture and the internal dependencies of an object-oriented system. Therefore a simple graphical model is introduced based on ideas from Coad & Yourdon (1991) and Yu & Adeli (1993).

1.3.1 Objects and classes

The behaviour of a system can be divided into a number of tasks. Each task is defined by some data that describe the current state and a number of operations which can alter the state of the system leading to a new task. This provides a well-defined and logical structure which helps to maintain overview of the system. The idea of the object-oriented analysis and programming is to divide a system into tasks that directly reflect the system concepts.

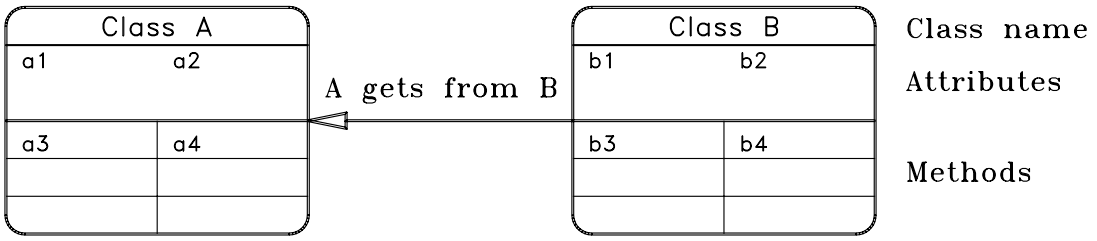


Figure 1.4: Classes and their dependency

In object-oriented analysis the behaviour of a system, e.g. a physical model, is described in terms of objects in a computer model. An *object* is an abstraction of a concept in the system, such as nodes and elements. It is described by a number of state variables called *attributes* which for a node may be label, coordinates, degrees-of-freedom, etc. The attributes are usually hidden from other parts of the model, i.e. they are *encapsulated*. The access to the attributes and behaviour of an object are modelled by *methods* which make it possible for other parts of the system to affect the state of the object, i.e. methods simulate the behaviour that the object is responsible for exhibiting, cf. Section 1.2. The base of an object-oriented architecture is the *class*. A class contains the description of one or more objects with a uniform set of attributes and methods, including a description of how to create a new object in the class, Coad & Yourdon (1991). Thus, an object is an initialized instance of a class, i.e. there exists the same relation as for type and variable in traditional compiled languages.

Objects communicate through *messages*. A message is a request to the object to alter its internal state, e.g. by performing an operation that alter the values of its attributes or by sending messages to other objects. Messages are the object-oriented counterpart to parameter lists in procedural programming. These can be avoided in object-oriented programming because an object itself knows how to react, i.e. to obtain and alter the involved attributes. E.g. in the calculation of the element stiffness the element will itself ask the material object to send the constitutive matrix. This means that the element depends on the material object. The dependency can act one way as for the element-material relation or it may be mutual as e.g. for a node-element relation.

In Figure 1.4 2 classes are represented graphically defining the class names, attributes and methods. The one-way dependency, Class A uses Class B, indicates that Class B by request sends information to Class A.

1.3.2 Inheritance

Classes alone do not make an object-oriented architecture - without *inheritance* a class would simply be an advanced version of a user-defined data structure. Inheritance is used to specialize the behaviour of a class. Instead of redefining the entire class a *subclass* is derived from the *superclass*, as e.g. an isoparametric element may be derived from an element superclass. The subclass inherits its attributes and methods from the superclass and new attributes and methods can be added. Those methods of the superclass that do not apply any longer can be redefined in order to specify a slightly different behaviour of

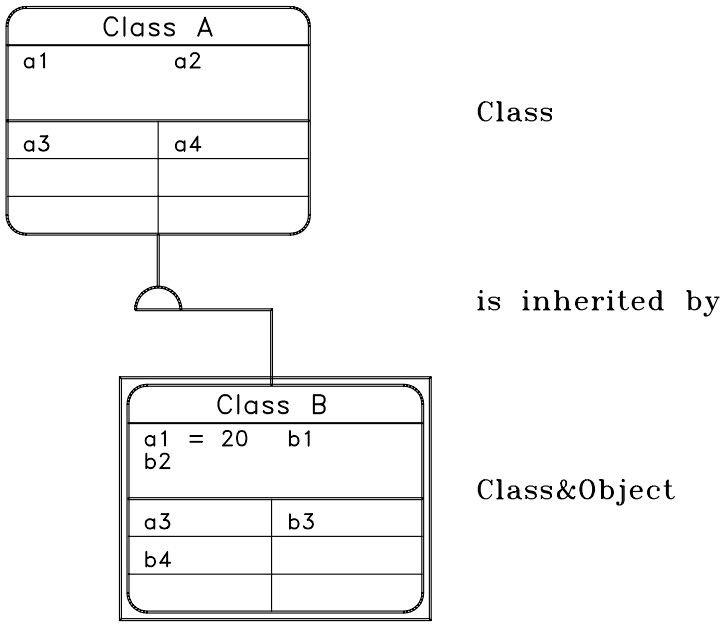


Figure 1.5: Inheritance - a class hierarchy

the subclass - elements would for example have to define the stiffness matrix. The subclass defines a class of objects, but it also initializes some of the attributes. The subclass is therefore at the same time a class and an object and will be referred to as a Class&Object. A system of classes which inherit from each other is referred to as a hierarchy. Graphically a class hierarchy is presented as in Figure 1.5. Attributes that are initialized by the subclass, e.g. `a1` and methods that are given new implementation by the subclass, e.g. `a3`, are defined both by the superclass and the subclass.

1.3.3 Polymorphism and dynamic binding

In order to implement an object-oriented architecture special features must be available in the programming language. Inheritance allows the subclass to use some of the methods of the superclass while others have a new definition and new methods are added. Objects of a class hierarchy share some methods but these may have different implementation. Shared methods are called *polymorph* - multiple shape - which means that a single declaration may apply to more than one implementation. The distinction between these relies on a strong typecasting and enables the program at run-time to seek the appropriate implementation of a method. Binding at run-time is called *dynamic binding*. First, the program will see if the object itself contains an implementation of the method, otherwise the superclasses are called in turn until an implementation is encountered.

1.3.4 Objects and C++

As discussed in Section 1.2 finite element programs consist of a mixture of data, methods and algorithms. The data and methods can conveniently be gathered in objects, whereas

the algorithms are more independent on the representation of the data. The objects are used for managing the model and developing a distributed program structure, while experience has shown that the procedural programming style is more efficient for numerical algorithms. A hybrid of procedural and object-oriented programming is therefore advantageous and will in many cases avoid the loss of efficiency often encountered for pure object-oriented applications, Forde et al. (1990). This is attractive for environments where highly specialized and optimized parts of scientific code are already implemented in C, Pascal or Fortran. C++ is a hybrid language where a procedural part - basically the C language subset - is extended to include object-oriented techniques. It is unlike Smalltalk, CLOS and Ada not entirely object-oriented, consequently the programmer is able to apply a programming style where part of the program is object-oriented - typically the data management part - while for the numerical part a procedural style as in C should be chosen.

In the following a short introduction to object-oriented programming in C++ will be presented. For a comprehensive introduction the reader should consult *The C++ Programming Language* by Stroustrup (1991), but also Lippman (1989) and Winder (1991) could serve as introduction. For the procedural programming part Kernighan & Ritchie (1991) gives an excellent introduction to C supplemented with Press et al. (1988) who present the basics of scientific programming in C.

The C++ declaration of objects is a class. A class is a user-defined type that can be used as any built-in type like integer or floating points. A class declaration consists of two parts. A *private* part which contains the attributes and some internal methods - this is the C++ counterpart to encapsulation. The *public* part is the interface to the other parts of the program. The public methods can manipulate the private attributes, i.e. calls to public methods correspond to sending a message to an object. Methods may be called with arguments, thus they are a mixture of pure object-oriented methods and procedures. The public part also consists of methods that are used to initialize new objects, these are called *constructors*. A constructor is typically responsible for memory allocation and initialization of attributes. When the object is not used any more a *destructor* is automatically called freeing the memory allocated for the object.

One of the reasons for the success of C++ is that it is a compiled language. For programs in C++ internal dependencies must be declared explicitly at compiling time, which allows the compiler to optimize these dependencies and produce a more efficient code. The drawback of this is that inheritance must be handled in a special way. In C++ inheritance and polymorphism are handled by *virtual methods*. The declaration of a virtual method tells the program to use dynamic binding for this specific method, i.e. previous implementations will be discarded. The reason that dynamic binding is not default is due to the fact that dynamic binding gives a slower code, while static binding (binding defined at compiling time) generally is faster. Static methods are also inherited but can not be overwritten. Static methods should therefore be used in cases where all versions of a method is identical for all subclasses. Consequently, the programmer should use as few virtual functions as possible, which actually is advantageous as it gives a small standard interface that is easy to overview. To make the derived classes work as fast as possible an extra level of encapsulation has been introduced - the *protected* attributes and methods of the superclass. These are shielded from the rest of the system like privates, but can

be accessed directly by the subclasses, i.e. the subclass can manipulate the shared data without sending the slower messages to the superclass. This is beneficial for the efficiency of the object-oriented code.

1.4 Review of literature

In this section a brief review of papers on object-oriented finite elements will be presented. Being a relatively new field there are few, but central papers notably by Baugh & Rehak (1989,1992), Forde et al. (1990) and Zimmermann et al. (1992). Class names will be indicated by a capital first letter, e.g. Element.

Baugh & Rehak (1989,1992) define an object-oriented framework for finite element analysis based on a geometric model described in terms of a Vertex class and an Edge class. The system consists of 3 FEM classes: Element, Node and Material. The Element class is defined by its topology, the material model, a type (e.g. isoparametric) and geometric parameters such as area, thickness etc. The topology information is inherited from the Edge class. The Material is itself a FEM class; it contains the constitutive properties of the model. The different types of elements (isoparametric, bar) are subclasses of the Element and provides an implementation of the element stiffness matrix. The last FEM class is the Node class which inherits its coordinates and connectivity (which edges are attached to this vertex) from the Vertex class. The Node also contains the degrees-of-freedom and information about boundary conditions (prescribed displacements and point loads). The analysis is controlled by a user application. The application is responsible for storing both the geometric and the FEM model. Having done that the model should be generated and the equation systems solved. The result - the unknown degrees-of-freedom - is stored by the Nodes. Any appropriate postprocessing such as stress evaluation may then be performed, it is, however, not an integral part of the Element class. The system was implemented in Common Lisp Object System (CLOS). Using a Lisp dialect is a natural choice for systems developed in a CAD environment, a fact used also by Miller (1991) who follows a similar approach.

Forde et al. (1990) presented an object-oriented finite element program for linear elastic analysis with plane, isoparametric elements. The aim was to develop an expandable framework that others could easily expand to more advanced problems or incorporate in expert systems. There are 5 FEM classes that have only a few attributes and methods. In addition to the Element, Material and Node classes, the boundary conditions are handled by a DispBC class and a ForceBC class. To each FEM class belongs a customized version of a List class which handles storage and assembly of the model. The Element is capable of computing the element stiffness and several types of distributed loads. This is done by numerical integration and requires a Gausspoint class and a Shapefcn class. Furthermore, the Element has its own postprocessing facilities such as stress evaluation and graphical presentation of the result. The entire finite element model is represented by a Domain class which stores the customized lists of Nodes, Elements, Materials and boundary conditions. It is also responsible for the storage of the global matrices and vectors. To perform an analysis the user should provide an application program that controls program evaluation,

i.e. definition of a Domain, solution and call of postprocessing facilities. The expandable framework is simple due to the small FEM classes. However, by giving the lists functionality that refers to a specific class instead of leaving that to the class itself the number of specialized classes becomes twice the necessary. The program was implemented in a hybrid language using C for the numerical part and ObjectPascal for the object-oriented part. A C++ implementation was provided by Scholtz (1992).

Zimmermann et al. (1992), Dubois-Pèlerin et al. (1992) developed a prototype program in Smalltalk in order to investigate whether object-oriented programming was applicable to finite elements. Their approach was guided by practical experience with FEM which is clearly reflected in the class structure. They introduce two levels of programming - the Domain class and the FEM classes. The Domain is responsible for managing the global model and for the solution of the problem. The 4 FEM classes are Element, Node, Material and Load. The Element is responsible for computing the stiffness and mass matrices and assembling them into the global arrays, computation and assembly of distributed loads, which are described by the Load class. Integration of the element matrices and vectors is done numerically. This requires a Gausspoint class, which in linear analysis is trivial, but becomes more important in material non-linear problems, Menétrey & Zimmermann (1992). The Node class defines coordinates and manages the degrees-of-freedom as well as the nodal loads. The degrees-of-freedom is defined in a separate Dof class, which stores the value of a dof and knows whether it is restrained or not. The Material is responsible for the constitutive behaviour of the model, thus it is used by the Element both in computing the stiffness matrix and calculation of the stresses. A C++ implementation was presented by Dubois-Pèlerin & Zimmermann (1993) which today is the most advanced version of object-oriented finite elements known to the author containing both static and dynamic analysis features as well as non-linear material models in addition to the standard linear ones.

Mackie (1992) uses ObjectPascal to test the possibility of changing from procedural programming to object-oriented programming. A class of Elements for plane stress and plane bending have been defined for static as well as dynamic analysis. The class methods are defined parametric in a style that lies close to the traditional Fortran style. What is accomplished by using object-oriented programming is, however, an enhancement of the program structure as well as reusability due to inheritance.

Yu & Adeli (1993) define a class library for finite element analysis. The analysis is centred around a GlobalElement object which handles the model assembly. It is a subclass of Element and uses several objects like Node, Material and Shape. The model is stored in a central database from which is is possible for any object to get the data that are needed. A notable difference to the systems presented above is the possibility for each object to copy itself, e.g. generate a number of equally spaced Nodes. The class library has been tested on composite laminate problems using a C++ implementation.

1.4.1 Matrix and vector classes

A major field within scientific programming is linear algebra which is used during the solution almost any engineering problem. The problems involve solution of linear equation

systems, multiplication of vectors and matrices, calculation vector products, etc. These tasks are trivial to the experienced programmer, still - or rather consequently - these parts of a program are error-prone, e.g. because of unbounded loops, uninitialized counters or wrong combination of indices. Therefore, a more symbolic programming style where loops and counters are replaced by symbols matching their mathematical interpretation is clearly of interest. This would lead to increased readability, fewer errors and increase the programmer's efficiency.

The class concept in object-oriented programming enables symbolic programming by use of overloaded operators. An overloaded operator is a redefinition of an ordinary operator to fit the current context. Several types of variables are relevant for consideration, especially mathematical types like rational numbers, complex numbers, vectors and matrices. In these cases a well-established symbolic notation exists that is similar to that of the simple types (integers and reals). Consequently, operator overloading makes it possible to create a more direct link between the mathematical concepts and their use in scientific programming. Classes such as Matrix and Vector can be used as tools in all types of programming. They are essentially not hierarchical in the classical object-oriented sense, e.g. an IntegerVector may not be derived from a RealVector. Such classes are termed abstract data types (ADT), Stroustrup (1991).

Several papers consider the possibility of using Matrix and Vector classes (in C++). Most authors use these classes as a tool for programming in a fully object-oriented framework, see e.g. Scholtz (1992), Dubois-Pèlerin et al. (1992), Yu & Adeli (1993). Others consider the influence that these classes may have on the development of procedural, scientific programming, Ross et al. (1992a,1992b), Nielsen (1993), Hededal (1993). It is concluded that using algebraic tools will enhance the development efficiency and make the code less error-prone. Also, it is found that the readability is improved due to the more symbolic programming style. The matrix and vector classes used in this work are presented in Appendix A.

1.5 Notation

Throughout this thesis a common notation will be kept in order to indicate objects, classes, attributes and methods. In order to distinguish between the text and references to object-oriented concepts these will all be written with sans serif typeface. By tradition class names and objects of a class are spelled with capital first letter, e.g. Element, Node, Material. For attributes and methods small letters will be used. Names of methods contain a verb indicating the message sent to the object, e.g. put_label means that the object should return its identification attribute, label. In cases where it can be unclear to which class an attribute or method belongs it will be denoted by its class and name, e.g. Element.put_label. Programming details are illustrated in pseudo-code. The code segments are written in sans serif typeface and divided from the text by horizontal lines, i.e.

code segment

```
Node.put_label()
  return label
```

The pseudo-codes are not direct transcriptions of the C++ syntax.

In mathematical formulations scalar variables and functions are represented in standard italic typeface, e.g. x_i represents a coordinate in direction i. Subscripts generally refer to the spatial direction, while superscript usually refers to a node or element number. Matrices and vectors are written in boldface, e.g. $\mathbf{B}$ or $\boldsymbol{\sigma}$. The superscript, ^{T}, marks the transpose of a vector or matrix.

Chapter 2

Concepts in finite elements

In this chapter the finite element formulation of potential problems is described. The potential problem has the advantage that it is a scalar problem which is simple but still it possesses all characteristics of a finite element formulation. The object is to identify some key concepts that are common to all finite element formulations regardless if it is potential problems, solid mechanics or structural mechanics. Then a finite element formulation for elasticity theory is characterized by use of these concepts demonstrating the common structure in the method and pointing out the differences between the scalar and the vector problem. This leads to a unified notion for the finite element concepts and an identification of some parameters that may characterize the structure of the element matrices and vectors.

2.1 Balance equations

The basis of a finite element formulation in mechanics and mathematical physics is a balance equation on differential form and boundary and initial conditions, which in general is termed the strong form. For a potential problem as in Figure 2.1 it is the Poisson equation,

$$\boldsymbol{\nabla}^T \mathbf{q} + Q = 0 \tag{2.1}$$

where $\mathbf{q}^T = [\, q_1 \quad q_2 \quad q_3 \,]$ is the flux vector. $\boldsymbol{\nabla}^T = [\, \partial/\partial x_1 \quad \partial/\partial x_2 \quad \partial/\partial x_3 \,]$ is the divergence operator. The dimension of the flux vector and the divergence operator follows the spatial dimension. The scalar $Q = Q(\mathbf{x})$ is a load applied in point $\mathbf{x}$. Multiplication of (2.1) by a weight, w, and integration over the domain, Ω, yields,

$$\int_\Omega w \left(\boldsymbol{\nabla}^T \mathbf{q} + Q \right) d\Omega = 0 \tag{2.2}$$

which by use of the divergence theorem may be written as

$$\int_\Omega (\boldsymbol{\nabla} w)^T \mathbf{q} \, d\Omega = \int_\Omega w \, Q \, d\Omega \; + \int_\Gamma w \, q_n \, d\Gamma \tag{2.3}$$

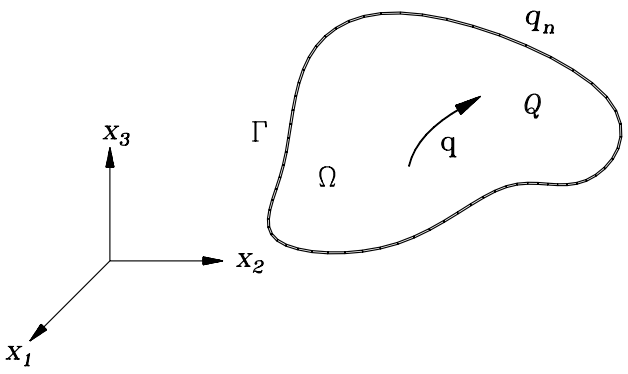


Figure 2.1: A potential problem

This is termed the weak form of the balance equation. It introduces the normal flux, $q_n = \mathbf{q}^T\mathbf{n}$, acting on the surface with the outward normal, $\mathbf{n}$. The gradient operator, $\boldsymbol{\nabla}$, which is adjoint to the divergence $\boldsymbol{\nabla}^T$, is defined as

$$\boldsymbol{\nabla} = \begin{bmatrix} \dfrac{\partial}{\partial x_1} \\ \dfrac{\partial}{\partial x_2} \\ \dfrac{\partial}{\partial x_3} \end{bmatrix} \tag{2.4}$$

The simple matrix relation between the divergence and the gradient in this case is due to the choice of a Cartesian coordinate system, but the adjoint pair can be determined in terms of other types of coordinate system. In this problem the divergence and gradient are the mathematical linear operators. The concepts, however, may be generalized to the matrix form encountered in elasticity theory or even introduced as non-linear operators in geometrically non-linear problems.

The weak form, (2.3), states that the internal forces (the left side) must balance the external loads (the right side). It is valid for all types of finite element formulations, both linear and non-linear. It is often assumed that there exists a linear relation between the flux and the gradient of the potential on the form

$$\mathbf{q} = -\mathbf{C}\boldsymbol{\nabla} u \tag{2.5}$$

where $\mathbf{C}$ is the constitutive matrix that for linear material models is constant. For non-linear material models, as for example plasticity theory, a linear constitutive relation is often used for the incremental formulation, e.g.

$$\delta\mathbf{q} = -\mathbf{C}\boldsymbol{\nabla}(\delta u) \tag{2.6}$$

Inserting (2.5) in (2.3) yields

$$\int_\Omega (\boldsymbol{\nabla} w)^T \, \mathbf{C} \, (\boldsymbol{\nabla} u) d\Omega = -\int_\Omega w \, Q \, d\Omega \; - \int_\Gamma w \, q_n \, d\Gamma \tag{2.7}$$

Table 2.1: Concepts in potential problems

Potential:	$u(\mathbf{x})$
Gradient:	$\boldsymbol{\nabla} u$
Flux:	$\mathbf{q}$
Divergence operator:	$\boldsymbol{\nabla}^T$
Gradient operator:	$\boldsymbol{\nabla}$
Internal forces:	$\int_\Omega (\boldsymbol{\nabla} w)^T \mathbf{q}\, d\Omega$
External loads:	$\int_\Omega w\, Q\, d\Omega + \int_\Gamma w\, q_n\, d\Gamma$
Constitutive models:	$\mathbf{q} = -\mathbf{C}\ \boldsymbol{\nabla}\mathrm{u}$

which is the weak form for potential problems with linear material models. This weak form has the potentials as unknowns, hence the flux may eventually be derived using (2.5).

The finite element formulation uses a number of basic concepts listed in Table 2.1. These concepts can be recovered in all types of finite element problems and they will in this thesis be used to obtain a unified finite element notion, cf. Section 2.4.

2.2 Finite element approximation

The basis of the finite element approximation is the weak forms of the balance equations, (2.3) or (2.7), depending on if it is a linear or non-linear problem that is considered. To be as general as possible (2.3) is used in the first place and then the linear formulation is recovered when introducing the element stiffness matrix.

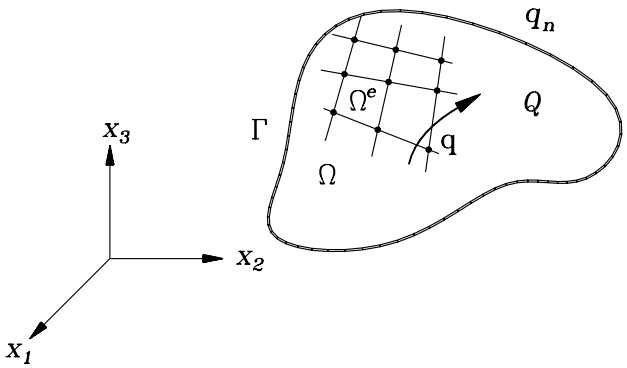


Figure 2.2: Discretizing in finite elements

The idea in the finite element method is to discretize the domain into a number of finite elements which are connected in the nodes as shown in Figure 2.2. The distributed

properties are lumped in terms of the nodal values - the degrees-of-freedom (dof). For the potential problem there is 1 dof in each node. To each dof corresponds a load, which is a sum of the external loads, i.e. of point loads acting in the node and a nodal equivalent to the distributed loads. The technique thus consists of reducing the problem from a continuous field problem to a discrete problem with a finite number of unknowns - the degrees-of-freedom.

Within each element the weight, w, can be approximated from the node values, $\mathbf{c}$, and the shape functions, $\mathbf{N}(\mathbf{x})$,

$$w(\mathbf{x}) = \mathbf{N}(\mathbf{x})\,\mathbf{c} \tag{2.8}$$

For an n node element with 1 dof in each node the shape function matrix, $\mathbf{N}$, has the form

$$\mathbf{N} = [\, N^1 \quad N^2 \quad \cdots \quad N^n \,] \tag{2.9}$$

where each shape function is defined so

$$N^i = \begin{cases} 1 & \text{in node } i \\ 0 & \text{in other nodes} \end{cases}$$

Introducing (2.8) into (2.3) and performing the integration over the element domain, Ω^e, yields

$$\mathbf{c}^T \int_{\Omega^e} (\boldsymbol{\nabla}\mathbf{N})^T\,\mathbf{q}\,d\Omega = \mathbf{c}^T \int_{\Omega^e} \mathbf{N}^T\,Q\,d\Omega \;+\; \mathbf{c}^T \int_{\Gamma^e} \mathbf{N}^T\,q_n\,d\Gamma \tag{2.10}$$

As $\mathbf{c}$ is independent of $\mathbf{x}$ it may be put outside the integrations. For an arbitrary choice of the weight the finite element approximation of the weak form finally becomes

$$\int_{\Omega^e} \mathbf{B}^T\,\mathbf{q}\,d\Omega = \int_{\Omega^e} \mathbf{N}^T\,Q\,d\Omega \;+ \int_{\Gamma^e} \mathbf{N}^T\,q_n\,d\Gamma \tag{2.11}$$

where the gradient matrix, $\mathbf{B} = \boldsymbol{\nabla}\mathbf{N}$, has been introduced as

$$\mathbf{B} = \begin{bmatrix} \partial N^1/\partial x_1 & \partial N^2/\partial x_1 & \cdots & \partial N^n/\partial x_1 \\ \partial N^1/\partial x_2 & \partial N^2/\partial x_2 & \cdots & \partial N^n/\partial x_2 \\ \partial N^1/\partial x_3 & \partial N^2/\partial x_3 & \cdots & \partial N^n/\partial x_3 \end{bmatrix} \tag{2.12}$$

The dimensions of $\mathbf{B}$ follows from the number of components in the gradient operator and the number of element nodes.

It is notable that the flux still has a general form and an approximation of the flux has not yet been defined. For linear material models the constitutive behaviour is modelled by (2.5), thus it is possible to approximate the flux in terms of the potentials. Within each element the potential field, $u(\mathbf{x})$, is approximated by the shape functions, $\mathbf{N}(\mathbf{x})$, and the node potentials, $\mathbf{a}^T = [\, a^1 \quad a^2 \quad \cdots \quad a^n \,]$,

$$u(\mathbf{x}) = \mathbf{N}(\mathbf{x})\,\mathbf{a} \tag{2.13}$$

Table 2.2: Finite element concepts

Element	
Node	
Shape function matrix:	$\mathbf{N}$
Gradient matrix:	$\mathbf{B} = \boldsymbol{\nabla}\mathbf{N}$
Element (tangent) stiffness matrix:	$\mathbf{K}^e$
Element load vectors:	$\mathbf{f}_v^e$, $\mathbf{f}_s^e$
Node load vector:	$\mathbf{f}_p^n$
Global degrees-of-freedom (dof):	$\mathbf{a}$
Global stiffness matrix:	$\mathbf{K}$
Global load vector:	$\mathbf{f}$

The shape functions are chosen to be the same as for the weight, i.e. the Galerkin approximation is used. Inserting this into (2.5) gives

$$\mathbf{q} = -\mathbf{C}\,\boldsymbol{\nabla}\mathbf{N}(\mathbf{x})\,\mathbf{a} = -\mathbf{C}\,\mathbf{B}(\mathbf{x})\,\mathbf{a} \tag{2.14}$$

whereby (2.11) becomes

$$\int_{\Omega^e} \mathbf{B}^T\mathbf{C}\,\mathbf{B}\,d\Omega\;\mathbf{a} = -\int_{\Omega^e} \mathbf{N}^T Q\,d\Omega\; - \int_{\Gamma^e} \mathbf{N}^T q_n\,d\Gamma \tag{2.15}$$

This recovers the finite element approximation of the weak form for linear problems, cf. (2.7). The integrals represents a system of n linear equations,

$$\mathbf{K}^e\,\mathbf{a} = \mathbf{f}_v^e + \mathbf{f}_s^e \tag{2.16}$$

$\mathbf{K}^e$ is the element stiffness matrix and is a $n \times n$ symmetric matrix. In non-linear cases it represents the tangent stiffness. The vectors $\mathbf{f}_v^e$ and $\mathbf{f}_s^e$ represent, respectively, the volume load and surface load acting on the element. The load vectors are the same for linear and non-linear models. Both the node potential vector, $\mathbf{a}$, and the load vectors have n components corresponding to the total number of dof in the element.

The relation (2.16) states the weak form of the balance equation for one element with approximated potential and weight. The global model receives contributions from all elements in the domain, thus the global stiffness matrix, $\mathbf{K}$, is found by assembly of the stiffness of all elements,

$$\mathbf{K} = \sum_{elements} \mathbf{K}^e \tag{2.17}$$

Likewise the element loads must be assembled into a global load vector, $\mathbf{f}$, which also receives contribution from point loads, $\mathbf{f}_p^n$, defined in the nodes,

$$\mathbf{f} = \sum_{elements} (\mathbf{f}_v^e + \mathbf{f}_s^e) + \sum_{nodes} \mathbf{f}_p^n \tag{2.18}$$

This enables restatement of (2.7) in a system of linear equations,

$$\mathbf{K}\,\mathbf{a} = \mathbf{f} \tag{2.19}$$

where $\mathbf{a}$ contains the nodal values of the potential. Solution of (2.19) will for linear models directly give an approximation of the potential field. For non-linear problems it represents the tangent behaviour of the system in the current state and may serve as the basis for an iterative solution strategy such as Newton-Raphson. This concludes the finite element approximation which has introduced a number of concepts listed in Table 2.2.

2.3 Elasticity theory

Elasticity theory differs from the scalar potential problem by being a vector problem where each dof - the displacement - is oriented. Typically, the dof is represented by a vector, $\mathbf{u}$, with components along each spatial direction, thus the displacement of each point is described by the same number of components as the spatial dimension. The formulation given below is for 3 dimensional elasticity, but apart from the matrix representations, the formulation is the same in 2 dimensions.

For the elastic body shown in Figure 2.3 the balance equation states equilibrium between the stress, $\boldsymbol{\sigma}$, and the volume loads, $\mathbf{b}$. It is for numerical purposes convenient to represent the original tensor relations in matrix format, i.e.

$$\boldsymbol{\nabla}^T\boldsymbol{\sigma} + \mathbf{b} = \mathbf{0} \tag{2.20}$$

where the stress vector is

$$\boldsymbol{\sigma}^T = [\,\sigma_{11} \quad \sigma_{22} \quad \sigma_{33} \quad \sigma_{23} \quad \sigma_{13} \quad \sigma_{12}\,] \tag{2.21}$$

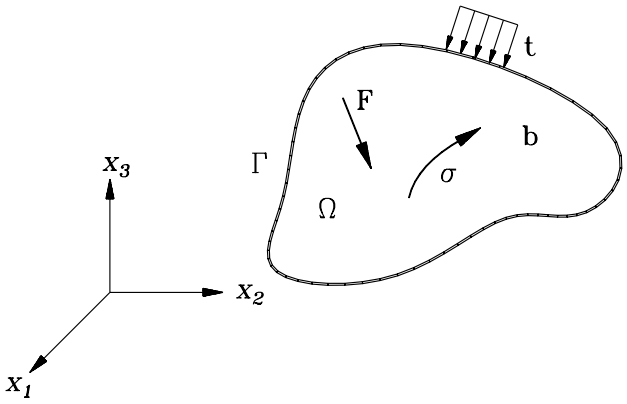


Figure 2.3: Elastic body

and the volume load is

$$\mathbf{b}^T = [\, b_1 \quad b_2 \quad b_3 \,] \tag{2.22}$$

For elastic problems the balance equation is formulated by use of a generalized divergence operator, $\boldsymbol{\nabla}^T$, which is on matrix form, viz.

$$\boldsymbol{\nabla}^T = \begin{bmatrix} \dfrac{\partial}{\partial x_1} & 0 & 0 & 0 & \dfrac{\partial}{\partial x_3} & \dfrac{\partial}{\partial x_2} \\ 0 & \dfrac{\partial}{\partial x_2} & 0 & \dfrac{\partial}{\partial x_3} & 0 & \dfrac{\partial}{\partial x_1} \\ 0 & 0 & \dfrac{\partial}{\partial x_3} & \dfrac{\partial}{\partial x_2} & \dfrac{\partial}{\partial x_1} & 0 \end{bmatrix} \tag{2.23}$$

Using the principle of virtual work the strong form is multiplied by a virtual displacement field, $\mathbf{w}^T = [\, w_1 \quad w_2 \quad w_3 \,]$, and integrated over the domain, Ω,

$$\int_\Omega \mathbf{w}^T (\boldsymbol{\nabla}^T \boldsymbol{\sigma} + \mathbf{b})\, d\Omega = 0 \tag{2.24}$$

The weak form is obtained by partial integration of (2.24)

$$\int_\Omega (\boldsymbol{\nabla}\mathbf{w})^T\, \boldsymbol{\sigma}\, d\Omega \;= \int_\Omega \mathbf{w}^T\, \mathbf{b}\, d\Omega \;+ \int_\Gamma \mathbf{w}^T\, \mathbf{t}\, d\Gamma \tag{2.25}$$

The partial integration leads to the generalized gradient operator, $\boldsymbol{\nabla}$, that is adjoint to the generalized divergence operator. In Cartesian coordinates the gradient matrix is the transpose of the divergence matrix, (2.23). The boundary term involves the traction, $\mathbf{t}$, which is the projection of the stress on the outward surface normal, $\mathbf{n}$, i.e. $\mathbf{t} = \boldsymbol{\sigma}^T\mathbf{n}$. The weak formulation thus has the same form as the potential formulation, (2.3).

For linear elastic materials the constitutive model, which expresses the relation between the displacements, $\mathbf{u}^T = [\, u_1 \quad u_2 \quad u_3 \,]$, and the stress, has the form

$$\boldsymbol{\sigma} = \mathbf{C}\boldsymbol{\nabla}\mathbf{u} = \mathbf{C}\boldsymbol{\varepsilon} \tag{2.26}$$

where $\mathbf{C}$ is the constitutive matrix. In this relation the strain, $\boldsymbol{\varepsilon}$, represents the gradient of the displacement vector. The strain, which is conjugate to the stress, is

$$\boldsymbol{\varepsilon}^T = [\, \varepsilon_{11} \quad \varepsilon_{22} \quad \varepsilon_{33} \quad 2\varepsilon_{23} \quad 2\varepsilon_{13} \quad 2\varepsilon_{12} \,] \tag{2.27}$$

Finally, the weak form of equilibrium for linear elastic bodies is found as

$$\int_\Omega (\boldsymbol{\nabla}\mathbf{w})^T\, \mathbf{C}\, (\boldsymbol{\nabla}\mathbf{u})\, d\Omega = \int_\Omega \mathbf{w}^T\, \mathbf{b}\, d\Omega \;+ \int_\Gamma \mathbf{w}^T\, \mathbf{t}\, d\Gamma \tag{2.28}$$

From this relation or the more general form, (2.25), a finite element approximation can be made as it is done for the potential problem. The displacement fields, $\mathbf{u}$ and $\mathbf{w}$, are

approximated by use of shape function matrix, $\mathbf{N}$, and the nodal displacements, $\mathbf{a}$ and $\mathbf{c}$. There are 3 displacement components in each node, thus $\mathbf{a}$ has the form

$$\mathbf{a}^T = [\, a_1^1 \quad a_2^1 \quad a_3^1 \quad \cdots \quad a_1^n \quad a_2^n \quad a_3^n \,] \tag{2.29}$$

This structure requires an expanded format of the shape function matrix,

$$\mathbf{N} = \begin{bmatrix} N^1 & 0 & 0 & \cdots & N^n & 0 & 0 \\ 0 & N^1 & 0 & \cdots & 0 & N^n & 0 \\ 0 & 0 & N^1 & \cdots & 0 & 0 & N^n \end{bmatrix} \tag{2.30}$$

Using the gradient operator on the approximated displacement field defines a block format of the gradient matrix, $\mathbf{B} = [\, \mathbf{B}^1 \quad \mathbf{B}^2 \quad \cdots \quad \mathbf{B}^n \,]$, where

$$\mathbf{B}^i = \begin{bmatrix} \partial N^i/\partial x_1 & 0 & 0 \\ 0 & \partial N^i/\partial x_2 & 0 \\ 0 & 0 & \partial N^i/\partial x_3 \\ 0 & \partial N^i/\partial x_3 & \partial N^i/\partial x_2 \\ \partial N^i/\partial x_3 & 0 & \partial N^i/\partial x_1 \\ \partial N^i/\partial x_2 & \partial N^i/\partial x_1 & 0 \end{bmatrix} \quad ; \quad i = 1, 2, ..., n \tag{2.31}$$

The gradient matrix has the dimensions $6 \times 3n$ where the number of rows corresponds to the number of strain components and the number of columns is equal to the number of dof times the number of nodes. The finite element formulation of linear elasticity theory then becomes

$$\int_{\Omega^e} \mathbf{B}^T \mathbf{C}\, \mathbf{B}\, d\Omega\ \mathbf{a} = \int_{\Omega^e} \mathbf{N}^T\, \mathbf{b}\, d\Omega\ + \int_{\Gamma_q^e} \mathbf{N}^T\, \mathbf{t}\, d\Gamma \tag{2.32}$$

identifying the element stiffness matrix,

$$\mathbf{K}^e = \int_{\Omega^e} \mathbf{B}^T \mathbf{C}\, \mathbf{B}\, d\Omega \tag{2.33}$$

as a $3n \times 3n$ symmetric matrix. Thus the dimension of the element stiffness matrix equal the number of dof times the number of nodes. The element load vectors

$$\mathbf{f}_v^e = \int_{\Omega^e} \mathbf{N}^T\, \mathbf{b}\, d\Omega \tag{2.34}$$

$$\mathbf{f}_s^e = \int_{\Gamma_q^e} \mathbf{N}^T\, \mathbf{t}\, d\Gamma \tag{2.35}$$

are of dimension $3n$.

Table 2.3 lists the elastic counterparts to the concepts already identified for the potential problem, cf. Table 2.1.

Table 2.3: Concepts in elasticity theory

Displacement:	$\mathbf{u}(\mathbf{x})$
Strain:	$\boldsymbol{\varepsilon} = \boldsymbol{\nabla}\mathbf{u}$
Stress:	$\boldsymbol{\sigma}$
Divergence operator:	$\boldsymbol{\nabla}^T$
Gradient operator:	$\boldsymbol{\nabla}$
Internal forces:	$\int_\Omega (\boldsymbol{\nabla}\mathbf{w})^T \boldsymbol{\sigma}\, d\Omega$
External loads:	$\int_\Omega \mathbf{w}^T \mathbf{b}\, d\Omega + \int_\Gamma \mathbf{w}^T \mathbf{t}\, d\Gamma$
Constitutive models:	$\boldsymbol{\sigma} = \mathbf{C}\,\boldsymbol{\varepsilon}$

2.4 Summary

In this chapter the finite element formulations of potential and elasticity problems have identified concepts that are shared by most finite element formulations.

The primary unknowns in a problem are the degrees-of-freedom (dof) that generally form a continuous field, but in the finite element formulation are discretized into a finite number of variables, $\mathbf{a}$, related to the nodes. Corresponding to the dof there exists an equivalent nodal load, $\mathbf{f}$, which comes from discretizing the external loads. A prescribed dof is referred to as fixed and to this corresponds an initially unknown load - a reaction.

The balance equation defines the generalized divergence operator, $\boldsymbol{\nabla}^T$. The adjoint generalized gradient operator, $\boldsymbol{\nabla}$, is introduced going from the strong to the weak form of the balance equation. In elasticity theory the generalized gradient of the displacement field is termed strain, $\boldsymbol{\varepsilon}$, this notion that will be used about the gradients in other problems as well. The strain is conjugate to the stress, $\boldsymbol{\sigma}$, which in the following will denote all types internal forces, e.g. flux or section forces. The relation between strain and stress is modelled by the constitutive matrix, $\mathbf{C}$, which depends on the material model.

A finite element approximation consists of discretizing the domain into elements which are connected in the nodes. Within each element the dof and weight are approximated by shape functions, $\mathbf{N}$, and thereby the approximation strain may be derived using the gradient matrix, $\mathbf{B}$. Inserting these approximations into the weak form, the element stiffness matrix, $\mathbf{K}^e$, and load vectors, $\mathbf{f}_v^e$ and $\mathbf{f}_s^e$, are obtained. The element contributions are assembled into the global stiffness matrix, $\mathbf{K}$, and the global load vector, $\mathbf{f}$. The load vector also receives contribution from prescribed point loads acting in the nodes, $\mathbf{f}_p^n$. The global dof vector, $\mathbf{a}$, contains initially of the values of the fixed dof which is defined in the nodes. This gives a linear system of equations whose unknowns are the free dof and the reactions corresponding to the fixed dof, thus the equation system is formed with 2 vectors that both can contain unknown values.

Table 2.4: Concepts in ObjectFEM

Concept	Symbol	Description
dof	$\mathbf{a}$	Primary unknown of the discretized problem
load	$\mathbf{f}$	Equivalent nodal load
fixed/free	-	A prescribed dof is fixed otherwise it is free
reaction	-	Nodal load introduced by a prescribed dof
strain	$\boldsymbol{\varepsilon}$	Generalized gradient of the dof field
stress	$\boldsymbol{\sigma}$	Internal force conjugate to the strain
stiffness matrix	$\mathbf{K}^e$, $\mathbf{K}$	Linear relation between dof and load
external load	$\mathbf{f}_v^e$, $\mathbf{f}_s^e$, $\mathbf{f}_p^n$	Applied loads
shape function	$\mathbf{N}$	Approximation of the dof field
gradient matrix	$\mathbf{B}$	Gradient operator for the discretized dof
constitutive matrix	$\mathbf{C}$	Relation between strain and stress
problem parameters	no_dof	Number of dof in each node
	no_strain	Number of strain components
element parameters	no_nodes	Number of element nodes
	no_gauss	Number of generalized Gauss points in each element

The main difference between different finite element formulations is the dimensions of the matrices and vectors that are involved. The problem formulation introduces 2 parameters: the number of dof in each node (no_dof) and the number of strain components (no_strain). The choice of element type introduces also 2 parameters: the number of nodes in the element (no_nodes) and the number of generalized Gauss points (no_gauss). The number of nodes may be different for each direction and should be defined as an array with the size of the spatial dimension. The Gauss points are used in the numerical integration of the stiffness matrix and the element loads, but are also the points where the strain and stress are evaluated from the dof. For elements with explicit integration, e.g. bars and beams, the number of Gauss points is simply the number of points where the strain and stress will be evaluated. Also the number of Gauss points may be different for different directions, thus should be defined as an array.

Table 2.4 summarizes the standard notion and symbols for the key concepts. These will be used in the following formulations and are also used by the FEM classes that form the object-oriented finite element program ObjectFEM.

The formulations given above have focused on linear problems. Still, relations such as (2.11) are general for both linear and non-linear problems. The main difference between the 2 problem types does not arise from the element formulation, but from the way the global equations are usually solved. While the linear equation system, (2.19), can be solved explicitly, the non-linear strategies iteratively seek a solution by solving a series of linear equation systems. These linear systems are stated on incremental form where the external load is applied in increments in a number load steps. The corresponding dof increments are adjusted by iterations so that the internal force is in equilibrium with the external load. In

each iteration the dof and may be the load are modified using the unbalance - the residual - between the internal and the external force and a linear predictor to calculate the new dof increment. The linear predictor generally represents an estimate to the tangent stiffness. The modifications that non-linear problems impose on the structure of the problem are thus limited to an incremental formulation of the balance equations which defines the evaluation of the internal force and replaces the stiffness with a tangent stiffness.

Chapter 3

Classes in finite elements

With the previous chapters as basis it is possible to define an object-oriented finite element framework called ObjectFEM. First, the structure of a program for linear analysis is shortly presented and the requirements that govern the management structure are discussed. Then follows the presentation of the FEM classes: Node, Element, Material and Property. There will be a description of their function in the framework and the key attributes and methods. The List class and the algebraic classes Matrix and Vector are then introduced. Finally, the application which defines the global model and describes a linear analysis is presented indicating the parts that must be modified when introducing new elements or materials or in order to solve non-linear problems.

3.1 The class structure

On basis of the finite element formulations in Chapter 2 it is possible to identify the 4 FEM classes in ObjectFEM: Node, Element, Material and Property.

The Node is a connection point for the elements and its responsibility is to manage the primary unknowns in the problem: dof and load. Each dof and its corresponding load form a dual pair, where either of them must be prescribed in order for the global equation system to be non-singular. This duality affects the solution of the global system, because the prescribed dof imposes an initially unknown load on the system, thus the global dof and load vectors both contain unknowns. In this thesis the duality is taken into account in the solution methods so that the Matrix.solve method simultaneously determines the unknown dof and reactions. The Node must be able to send prescribed dof and loads to the global equation system and retrieve the initially unknown values after the global analysis. The Node may also define a point in the physical domain represented by a set of coordinates. The distinction between element connection points and physical points could be exploited to introduce slave nodes in the elements. These slave nodes, which do not have coordinates, could be eliminated before assembling the global equation system as done for superelements.

The Element takes care of the distributed properties. Its tasks can be divided in preprocessing and postprocessing. During preprocessing the Element collects properties, such as geometry and material behaviour, from the other classes in order to process the discretized

forms, i.e. the element stiffness matrix and the element load vector, to be assembled into a global equation system. Postprocessing consists of interpreting a discrete solution represented by the dof in terms of distributed properties, namely the strain and stress, requiring again information from the other classes.

The Material describes material models which the Element can use in an analysis. Its task is to manage the material parameters so the Element at any time is able to evaluate the stiffness and the stress. For linear materials the constitutive matrix uniquely defines the material behaviour. The Property class is introduced to take care parameters that are not specifically part of the material models. For plane problems it is the thickness, while for bars and beams they are the cross section areas, moments of inertia, etc. These parameters are on request send to the Element.

The entire finite element model consists of objects of all 4 types. In addition to the finite element formulation there is a number of tasks which must be taken care of: model definition, model generation, forming the global equation system, solving the global equation system and postprocessing. These tasks are handled by the application. Zimmermann et al. (1992) introduce a domain class to control the analysis, but in ObjectFEM a user defined application module handles the tasks mentioned above.

Defining the model consists of reading and storing all the objects in the model and prescribing the boundary conditions. The model should be stored in a dynamic storage structure, like arrays, linked lists or trees, that is gradually expanded for each defined object, thus the model size needs not to be known from the beginning, e.g. allowing a step by step definition of elements and nodes in different parts of the physical domain. In the model generation phase the relations between the objects are determined, i.e. the Elements search the different stores to obtain references (addresses) to the connected Nodes, Material and Property and vice versa. Generating the model also enables determination of the structure global stiffness matrix - an initial dof numbering scheme may serve as a first guess on a numbering sequence. If the stiffness matrix structure is not convenient a renumbering scheme could be employed, see e.g. Schwarz (1988).

Each object in the model is now able to perform its tasks. The data that form the global equation system is divided in 2 types: prescribed data and processed data. The prescribed data are dof boundary conditions and point loads acting in the nodal points. These data are sent directly to the global equation system and thus do not require any type of processing. The processed data are the element contributions: stiffness and distributed loads. It is notable that the Material and Property objects are not directly used by the application, but the values of their attributes are processed by the Element and sent to the application. The global equation system, linear or non-linear, needs to be solved to obtain the unknown dof and node loads. For this purpose the application requires access to a solver for linear equations. Optimal solution algorithms depend on the structure of the matrix, whether it is banded, symmetric, a one- or two-dimensional array etc. Like for FEM attributes and FEM methods, it is convenient to relate the algebraic methods to the matrix and vectors attributes, thus defining Vector and Matrix classes. For non-linear problems there is also involved a strategy which is defined in the application.

Having obtained the unknown dof and reactions the results may be returned to the nodes. There the results may be processed into tables or graphics. The distributed results,

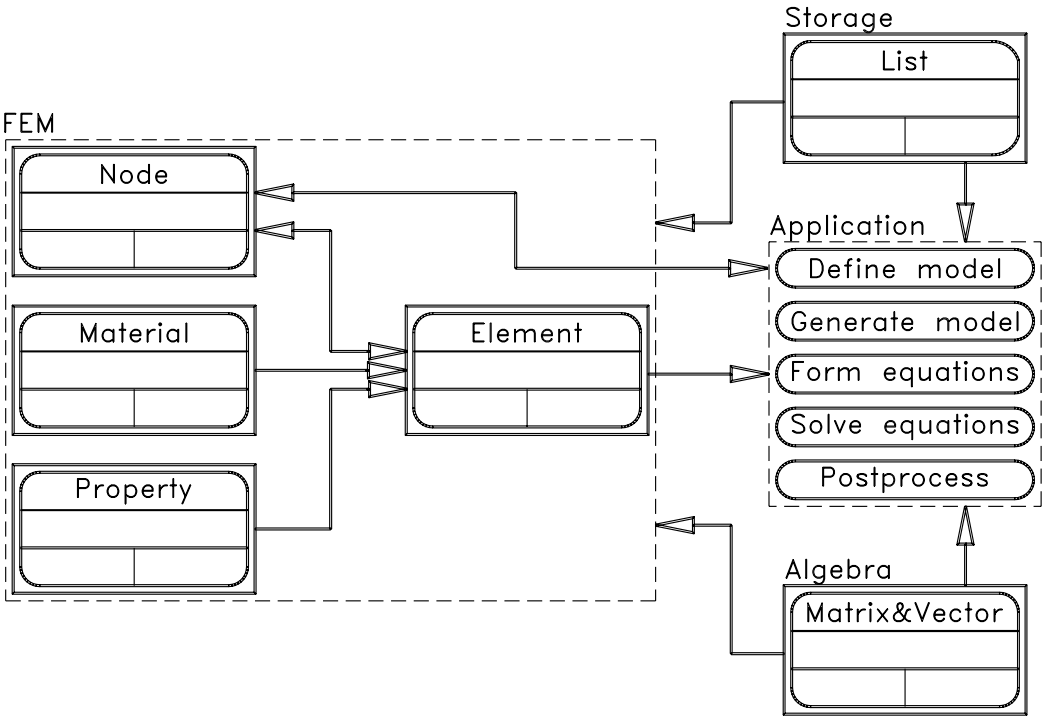


Figure 3.1: Structure of ObjectFEM

strain and stress, is handled by each element in turn, deriving the results from the dof.

The structure of a object-oriented finite element program is shown in Figure 3.1. A more detailed description of the different parts is given the following.

3.1.1 Requirements to ObjectFEM

The idea behind ObjectFEM is that all types of linear problems in principle can be treated by the same application, regardless the choice of elements and materials. This aim imposes certain requirements on the classes:

- Self-contained classes
- Standard class interface
- Groups of dof

First of all, the classes should be self-contained, i.e. they must be able to represent themselves in every part of the analysis performing I/O, model generation, and postprocessing. This will allow a very general definition of the linked list class that stores and manages the entire finite element model.

It must be possible to refer to an element in a standard way, e.g. isoparametric, thermal elements and beam elements should simply be referred to as elements. In order for this to work a standard interface must be defined through which all types of elements communicate with the global equation system. This standard interface must be defined by the superclass, Element, and inherited by its subclasses. The subclasses overwrite previous implementations of for example the stiffness matrix. On the other hand, there are methods

which are identical for all types of elements. For example, assembly of the global stiffness matrix employs a standard scheme for transferring element contributions into the right places in global matrix. It is important to realize the difference between the two types of methods when programming in C++. The first type of methods requires possibility of redefinition, for these dynamic binding is used by defining them as virtual methods. For the second type static binding is sufficient as these do not require new implementations by the subclasses. Instead, the static methods use the problem parameters and virtual methods which are set by the different subclasses.

A final requirement is the ability to mix different types of elements in an analysis. Analyzing a structure for which one part is subjected to combined elastic and thermal loads, while the other only is loaded elastically, it would be convenient to mix elements which include the thermal contribution with elements that do not. Assembling these correctly requires identification of the different types of degree-of-freedom, telling which part of the stiffness matrix refers the elastic dof and which part concerns the thermal dof. Standard numbering of dof will have to be introduced as e.g. done in ABAQUS (1992) where

1-3: Translations
4-6: Rotations
7: Warping
8: Pressure
⋮
11: Temperature

In ObjectFEM the different types of dof is ordered in groups. Presently, 3 groups of 3 dof each are defined:

Group 1: Translations
Group 2: Rotations
Group 3: Scalars - temperature, pressure, etc.

The groups will be used when communicating between objects, but internally the dof are stored in a single array.

3.2 The FEM classes

The finite element formulation is handled by the FEM classes: Node, Element, Material and Property. Attributes and methods are generally named according to the standard finite element notion introduced in Section 2.4. There are, however, some attributes and methods which do not have counterparts in the theoretical formulations. A guideline for the choice of names and the syntax is given in the following. It covers most of the attributes and methods, while a more detailed description of the key attributes and methods is given in Sections 3.2.1–3.2.4.

Pseudo-code representations are introduced to support the text. The syntax for class methods will be as follows

class methods

```
Class.method1(argument list):
   implementation...
```

The first line represents the declaration, which is identified by class name (capital first letter), method name and argument list. The remaining part gives the implementation of the method.

Control structures and calls to the methods of an object are represented as

call to method

```
for (i=1 to n) do
   object1.method1(argument list)
```

An object of a class, e.g. object1, is referred to in small first letter. Methods are referred to by their object name, method name and an argument list enclosed in parentheses, e.g. object1.method1(). Attributes are referred to by variable and attribute name (without parentheses), e.g.

access to attribute

```
object1.attribute1 = ···
```

shows assignment to attribute1 in object1.

Most of the methods are polymorph, i.e. they are used by more than one class and there generally exist several versions of the same method within each class. The difference between the versions are the number and type of arguments. Another important issue is the possibility to use the return value of a method as the left-hand side in an assignment operation, e.g.

assignment

```
elem.set_node(1) = "Node1"
```

assigns the string Node1 to Element.nodelabel(1). The Element method has the following form

set node

```
Element.set_node(i):
   return nodelabel(i)
```

Attributes

In this framework all objects are uniquely identified by their label, which generally is a character string. Attributes called element, node, material or property are used to store references (addresses) to such objects. The prefix no_ indicates that the attribute stores a number, e.g. Element.no_nodes is the number of element nodes. Values of model properties, such as Youngs modulus or the thickness of a plate, are stored in arrays called par.

Methods

The class attributes are encapsulated. Therefore there must be defined a set of methods that makes it possible to obtain the value of an attribute or assign a new value. These *access methods* will be characterized by 2 verbs: set and put. The set method assigns a value to an attribute, e.g. set_label instructs the object to assign the argument to the label attribute and node.set_coor assigns a coordinate vector to the attribute Node.coor. The methods also illustrate 2 ways of assignment: the first takes an argument, while the latter uses an overloaded assignment operator, Vector.operator =, i.e.

```
———————   set methods   ———————
elem.set_label("Element1")
node.set_coor() = point_A
```

The set methods usually set a single attribute, but are in some cases used for more complicated operations as it is demonstrated later on. The put method returns the value of the requested attribute, e.g. Node.put_coor returns the coordinates of a Node object. The method is polymorph and can be used with or without an argument, i.e.

```
———————   put methods   ———————
point_A = node.put_coor()
x_coor = node.put_coor(1)
```

The *I/O methods* are defined for communication with a terminal or a text file. The read methods read formatted input, e.g. Node.read reads the label and the coordinates and Element.read_vload reads the nodal intensities of the element volume load. Output is handled by a write method which either echoes the input directly, like Material.write that echoes the label and the material parameters, or presents the results of the analysis, e.g. Node.write_dof. The methods uses the terminal if a file is not given in the argument list.

```
———————   read and echo   ———————
node.read()
node.write()
elem.read(infile)
elem.write(outfile)
```

The *management methods* handle interchange of larger amounts of data. The methods that bring information from the object into the global equation system have send or assm as prefix depending on whether it is prescribed or processed data, e.g. Node.send_load introduces prescribed point loads in the global load vector, while Element.assm_load sends the processed values of the distributed loads to the global equation system. Retrieving information from other parts of the system is handled by get methods, e.g. Element.get_dof tells the Element object to retrieve the dof values from its Nodes. The methods that communicate with the global equation system require an argument such as the global stiffness matrix, while communication between the objects is handled without arguments, e.g.

Node	
label no_dof no_dofgroup dofgroup coor elements dofno dof load fix	
set/put label	set/put coor
set/put dof	set/put load
set element	set dofno
resize vectors	send/get dof
send/get load	read/write
read/write dof	read/write load

Figure 3.2: The Node class

management
elem.assm_stiffness(K)
node.get_load(f)
elem.get_dof()

The last type of methods are the *FEM methods* which relate directly to concepts in the formulation. They are used to establish the element stiffness matrix and the element load vectors and they define the strain and its conjugate stress. These methods do not have a verb as prefix, thereby obtaining a more symbolic programming style. They are presented along with the objects in the following chapters.

3.2.1 The Node class

The Node is the most general of the 4 FEM classes and is not inherited by any subclasses. It serves 2 purposes: collecting the dof and describing a point in the physical domain. The Node is presented in Figure 3.2 and descriptions of key attributes and methods are given below.

The Node is a connection point for elements. References to the connected Element objects are stored in elements. An arbitrary number of elements may be connected and therefore elements is defined as an ElementList which can be extended dynamically, see Section 3.3.1. The type of elements determines the type and number of dof. These are numbered by 3 attributes: dofgroup, no_dof and no_dofgroup. The dofgroup array stores the number of active dof in each of the 3 (=no_dofgroup) groups, cf. Section 3.1.1. Initially there are 3 available dof in each group, but during the model generation, the actual number is determined and stored in dofgroup. The total number of dof is stored in no_dof. Initially, no_dof=9 but it is reset after determining the actual number of active dof.

The Node have 3 vectors with dimension no_dof that describe the dof: dof, load and fix. Before the analysis the dof vector contains the prescribed dof values, i.e. the dof boundary conditions. The fix array is a boolean indicator that tells whether the dof is fixed or free. To each dof there corresponds a nodal load stored in the load vector. This vector stores prescribed point loads before the analysis is performed. After the global analysis the unknown dof and reactions are retrieved and stored the remaining positions in dof and

load. The position of nodal contributions to the global dof and load vectors is stored in the dofno array, whereby the Node itself is able to exchange information with the global equation system.

The Node may also describe a point in the physical domain. The point is described in terms of coordinates, coor, referring to the global coordinate system. The coordinates are used by the Elements to carry out the integrations of the stiffness and distributed loads. As an option the Node could define a local coordinate system - this is not implemented in ObjectFEM.

The boundary conditions may be prescribed using set_dof and set_load and retrieved by their corresponding put methods. The set_dof method illustrates the principle. Prescribing a dof involves also setting a position in the fix array, i.e.

set dof

```
Node.set_dof(dg,dn):
   index = no_dofgroup * (dg–1) + dn
   fix(index) = TRUE
   return dof(index)
```

where dg refers to the dof group (1,2,3) and dn is a local dof number (1,2,3), that internally is translated to a one-dimensional numbering scheme. The value of the dof is set by an assignment operation, e.g.

prescribe dof

```
node.set_dof(1,2) = 0
```

fixes translation 2. It should be noticed that the methods during the model definition assume that all 9 available dof are active, thus if it is later encountered that a prescribed dof is not active it will simply be discarded when compressing the internal vectors, see resize_vectors.

Forming the global equation system requires the global dof numbers to be set. The number of active dof in a Node depends what types of elements that are connected. During the model generation the Elements send information about how many dof are active in each dof group. This is done through the set_element method, which stores the number of active dof in dofgroup and a reference to the calling Element in elements.

set element

```
Node.set_element(elem):
   elements.add(elem)
   for (i=1 to no_dofgroup) do
      dofgroup(i) = max(dofgroup(i),elem.get_dofgroup(i))
```

The attribute elements is an ElementList, which can be extended dynamically by use of the add method, cf. Section 3.3.1.

Having determined the number of active dof in all Nodes the global dof number, dofno, may be set using set_dofno, which assigns a unique number to all dof in the model. The Node sends a message about dofno to each connected Element.

set dofno

```
Node.set_dofno(gl_dofno):
  for (i=1 to no_dofgroup) do
    no_dof += dofgroup(i)
  for (i=1 to no_dof) do
    gl_dofno += 1
    dofno(i) = gl_dofno
  k = 0
  for (el=elements.start() to elements.end) do
    for (i=1 to no_dofgroup) do
      for (j=1 to dof_group(i)) do
        k += 1
        elem.set_dofno(label,i,j) = dofno(k)
  resize_vectors()
```

The method takes the previous global dof number, gl_dofno, as argument and sends the incremented global dof number back to the system. The operation k += 1 is a compact notation for k = k+1. At this moment of the analysis the number of active dof in the node is known and in order to minimize the storage used by the object the internal vectors are resized before allocating the global system. This is done by the resize_vectors method, which sets the size of the internal vectors equal to the number of active dof (=no_dof).

resize vectors

```
Node.resize_vectors():
  Vector dof1(no_dof)
  k = 0
  for (i=1 to no_dofgroup) do
    for (j=1 to dofgroup(i)) do
      k += 1
      l = no_dofgroup * (i–1)+j
      dof1(k) = dof(l)
  dof = dof1
```

The method includes also resizing of the load and fix vectors, but being identical they are omitted here.

The discrete boundary conditions enter without modifications into the global equation system. The Node has 2 methods for communicating with the global system: send_dof and send_load. These methods use the global dof number, dofno, to place the prescribed values correctly to the global vectors. A global boolean array gl_fix marking the prescribed dof is formed along with the global dof vector by send_dof.

send dof

```
Node.send_dof(a,gl_fix):
  for (i=1 to no_dof) do
    if (fix(i)=TRUE)
      a(dofno(i)) = dof(i)
      gl_fix(dofno(i)) = TRUE
```

The load is send to the global system by a similar method.

Element	
label no_dofgroup dofgroup dofno no_dof no_strain no_nodes no_gauss nodelabel matlabel proplabel nodes material property vload no_surf no_snodes sload	
set/put label	set/put node
set/put mat	set/put prop
generate model	set dofno
assm stiffness	assm load
get dof	
stiffness	load
strain	stress
read/write	read/write vload
read/write sload	
write strain	write stress

Figure 3.3: The Element class

After solving the constrained system of equations, $\mathbf{K}\,\mathbf{a} = \mathbf{f}$, the Node may retrieve the unknown dof and reactions from the global vectors by use of the get_dof and get_load. As for the send methods the global dof numbers are used to access the vectors correctly, e.g.

get dof

```
Node.get_dof(a):
   for (i=1 to no_dof) do
      if (fix(i)=FALSE)
         dof(i) = a(dofno(i))
```

Notice that only the unknown values are extracted.

3.2.2 The Element class

The Element is a superclass for all types of elements. It implements the access, I/O and management methods. The FEM methods stiffness, load, strain and stress are declared but not implemented (purely virtual methods) and a subclass should provide implementations of these methods. A general finite element formulation introduces shape functions and a gradient matrix that are used when evaluating the stiffness, load and strain. Some elements, like for example bars and beams, have explicit forms of the stiffness matrix and the strain and it is therefore chosen not to include such methods in the superclass. The class is presented in Figure 3.3.

An Element is described by 2 problem parameters: no_dof, no_strain and 2 element parameters: no_nodes and no_gauss. The number of nodes and dof in each node define the size of the element arrays, matrices and vectors and are used to control the I/O and management methods. These methods, e.g. set_dofno, assm_stiffness and assm_load, thereby become identical for all elements and may be defined statically. The no_strain sets the

number of strain (and stress) components which are calculated in no_gauss points within the element, typically the Gauss points. The write_strain and write_stress can thus also be defined statically. To communicate with the Nodes the dof are divided into 3 dof groups. As the Element is defined by its dof, it is responsible for setting dofgroup upon initialization.

All the listed parameters are specific for each element type and therefore these are set by the subclass. For an 8 node potential element in 3D, (Pot3D8), the constructor may look as follows.

Pot3D8

```
Pot3D8.constructor:
   no_dof = 1
   no_strain = 3
   no_nodes = 8
   no_gauss = 8
   dofgroup(3) = 1
```

The topology is described in 2 tempi. In the model definition phase labels that refer to yet unknown objects are stored in nodelabel, matlabel and proplabel. During model generation these labels are translated to references to actual objects which then are stored in the nodes array, material and property. The Element knows the size of the nodes array, thus it is not necessary to use a NodeList.

The nodal intensities of the volume load, vload, and surface loads, sload, are defined by each Element separately. The volume load, vload, is a vector with no_nodes×no_dof components. The surface load must be defined for each surface and therefore 2 additional parameters, no_surf and no_snodes, are introduced to set the size of the matrix, sload. This structure should in the future be revised introducing a LoadBC class, which can describe the different types of load, see e.g. Forde et al. (1990). The Element should then store a reference to a LoadBC object which can be called during the calculation of the equivalent nodal loads.

The key method in ObjectFEM is generate_model which creates references between the objects in the model. The element topology is at first defined in terms of nodelabel, matlabel and proplabel. The method generate_model searches the NodeList, MaterialList and PropertyList to obtain references to the objects and stores these in nodes, material and property. This enables the Element to collect the data used in calculation of the stiffness matrix, the load vector etc.

generate model

```
Element.generate model(nolist, matlist, prolist):
   for (i=1 to no_nodes) do
      node(i) = nolist.find(nodelabel(i))
      node(i).set_element(this)
   material = matlist.find(matlabel)
   property = prolist.find(proplabel)
```

In the process Node.set_element is called to create a mutual reference. The argument this is the way that an object refers to itself.

After the generation of the model it is possible to define the structure of the stiffness

matrix and by use of Node.set_dofno to define the global dof numbers. In order for the Element to be able to assemble the stiffness and loads correctly a corresponding method is defined for the Element,

set dofno

```
Element.set_dofno(nlabel,dg,dn):
  for (i=1 to no_nodes) do
    if (nodelabel(i) = nlabel)
      index = (i–1) * no_dof
      for (j=1 to dg) do
        index += dofgroup(j)
      return dofno(index+dn)
```

The method takes a node label (nlabel), the number of the dofgroup (dg) and the local dof number dn as arguments. After seeking the correct element dof the dofno is set by assignment. Instead of finding the correct entry by looping over all preceeding dof groups an address array could be formed containing e.g. the number of the first dof in each group, thus enabling a more direct notation.

The Element is responsible for calculating and assembling the processed part of the global model. The virtual methods, stiffness and load, are called by the static assembly methods, assm_stiffness and assm_load. By use of the global dof numbers, dofno, each contribution is then added to the global equation system. Calculation and assembly of the stiffness may illustrate the principle,

assm stiffness

```
Element.assm_stiffness(K):
  Ke = stiffness()
  size = no_dof * no_nodes
  for (i=1 to size) do
    for (j=1 to size) do
      K(dofno(i),dofno(j)) += Ke(i,j)
```

It is seen that there is a direct relation between the global dof numbers, dofno(i), and the local dof numbers, i.

Having solved the global equation system the unknown node dof and loads are retrieved by the Nodes. In order for the Element to evaluate the strain it needs to extract the dof from the connected nodes. This is handled by the get_dof method.

get dof

```
Element.get_dof():
  l = 0
  for (i=1 to no_nodes) do
    for (j=1 to no_dofgroup) do
      for (k=1 to dofgroup(j)) do
        l += 1
        dof(l) = node(i).put_dof(j,k)
```

Once again it is seen that internally the dof are numbered consecutively while communi-

Material	
label	no_par par
set/put label	set/put par
C	read/write

Figure 3.4: The Material class

cation between different objects uses the dofgroup number and a local dof number.

The postprocessing facilities are limited to calculation of the strain and stress. The strain and stress definitions are handled by virtual methods. The Element, however, implements a write method for writing a table of strain and stress for no_gauss specified points in the element. In principle this may look the following way,

write strain

```
Element:write_strain()
   E = strain()
   write(tablehead)
   for (i=1 to no_gauss) do
      write(i)
      for (j=1 to no_strain) do
         write(E(i,j))
```

It is assumed that strain returns a matrix with the correct dimensions.

3.2.3 The Material class

The Material is the superclass for all material models. It introduces the constitutive model described by no_par parameters. The parameters are stored in the par vector. On request it provides the element with a single parameter,

put par

```
Material.put_par(pn):
   return par(pn)
```

The full constitutive matrix is used to evaluate the stiffness and the stress. This is handled by the purely virtual method, C, which must be implemented for each type of material. The class, which is shown in Figure 3.4, will increase in importance for material non-linear problems such as plasticity theory.

Property	
label	no_par par
set/put label	set/put par
read/write	

Figure 3.5: The Property class

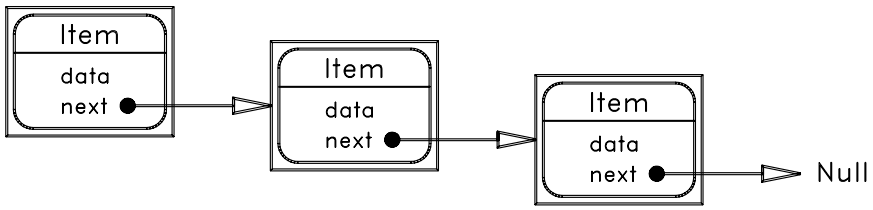


Figure 3.6: A linked list

3.2.4 The Property class

The Property class is a simple class which is described by the number of parameters, no_par. It is managing parameters that are not specifically part of the material model. E.g. for plane strain or stress one parameter will be the thickness of the element, while for bars and beams the area and moment of inertia are to be defined. The class will on request from the Element return one of the parameters stored in par, see Material.put_par.

3.3 Model storage

Like the rest of the system the storage model must be designed to be as flexible as possible. The flexibility is obtained by definition of dynamic storage schemes like arrays, linked list, trees or even databases. Arrays are generally easy to handle and are widely used to store finite element models, but in order to perform range checking the size of the array must be known from the beginning. Gradual extension is easier to obtain with linked lists where range checking is replaced by an existence check indicating the end of the list. A linked list requires more memory to store the same number of items than an array. This drawback is, however, compensated by the possibility to add or remove items dynamically which can be used for example in free-meshing techniques or adaptive methods where the mesh is automatically refined. ObjectFEM uses linked lists to store the FEM objects in the model.

3.3.1 The List class

A linked list consists of Items. An Item has 2 fields: data and next. The data field can be any type of variable, e.g. integer, floating point, arrays, data structures or objects. The next field holds the address of the next Item in the list. Extension of the list consists of allocating the storage for a new data field and setting next to point at an empty field - the Null pointer. The new Item is inserted in the end of the list replacing the next field of the previous Item with the address of the new Item. The end of the list is thus indicated letting next point at an empty field. The principle of linked lists is shown in Figure 3.6.

2 item variables are needed for managing the list: the first item in the list, start_item, and the current item, curr_item. Running through the list thus consists of looking in the next field of the current item to find the address of the following item. Safe and efficient manipulation of a linked list can be obtained by making it a class. The List is designed as template class, i.e. a class that can be typecasted to any data type without modifications.

A List consists of an arbitrary number of Items each pointing at the next one in the list. The merit of the List class is to deliver methods that enable safe and easy manipulations of these Items. The List should be able to add, remove or find an Item and there should be a method, next, that steps through the List returning the data. This is necessary in cases where a message should be send to all Items in the List. The List is represented in Figure 3.7.

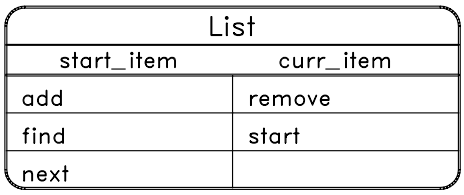


Figure 3.7: The List class

3.3.2 Customizing the lists

All objects in the finite element model are stored in a linked list that is typecasted from the template, List. There are defined a NodeList, an ElementList, a MaterialList and a PropertyList. For these the data are addresses to objects of the type and the methods of e.g. the Element can be applied to items in the ElementList. This enables the List to simulate the traditional looping techniques such as the `for-do` and the `while-do` loops - a style that will enhance the readability e.g. when calculating and assembling the stiffness matrix. Assembly of the global stiffness by looping over each Element object (elem) in the ElementList (ellist) may be written in pseudo-code in the following way

for–do

```
for (elem=ellist.start() to ellist.end) do
   elem.assm_stiffness(K)
```

where start returns the first item in the list. The notion ellist.end is not an actual method, but it is introduced to retain the pseudo-code notation of the `for-do` loop. The representation above is a compact form of a `while-do` loop,

while–do

```
elem = ellist.start()
while (elem) do
   elem.assm_stiffness(K)
   elem = ellist.next
```

where the loop terminates when elem becomes Null. The use of the `for-do` loop is based on the possibility to simulate this syntax in C++ by use of an overloaded increment operator (++), i.e.

for-do in C++

```
for (elem=ellist.start(); elem; elem=ellist++)
  elem->assm_stiffness(K);
```

The more compact notation is therefore used in the pseudo-codes.

3.4 Algebraic classes

The major numerical part of a finite element program concerns linear algebra, i.e. matrix manipulations and calculation of vector products. For this purpose two types of classes are defined: Matrix and Vector. The classes are described by their body, m or v, and their dimensions, no_row and no_col.

Standard matrix algebra, like addition, substraction and multiplication, are defined as overloaded operators. This enables a symbolic programming style where loops are replaced by operators that match the mathematical concepts, e.g.

loop

```
for (i=1 to no_row) do
   for (j=1 to no_col) do
      A(i,j) = B(i,j) + C(i,j)
```

is replaced by

operator

```
A = B + C
```

The operators + and = have in this case been overloaded so they perform addition and assignment internally. Storing the dimensions allows the methods to check the validity of the operations, e.g. to ensure that the dimensions match or that the operations are within the matrix range. Accessing a member in a matrix is done by use of the overloaded index operator () that includes a range check,

index operator

```
Matrix A(2,2)
A(1,2) = 2
A(2,3) = 4
```

A is declared as a 2×2 matrix, so the last operation is illegal and therefore the assignment is not carried out.

The transpose of a matrix is symbolically represented by the method T, e.g.

transpose

```
A = L * D * L.T
```

reestablishing A from the matrices L and D which are the $\mathbf{LDL}^T$ factors.

The Vector class consists of a one-dimensional array. Its methods are divided between matrix algebra and vector calculus. The matrix algebra is similar to that of Matrix. The vector calculus part consists of methods for evaluating the inner (scalar) product, dot, the cross product, cross, and the Euclidian norm, length.

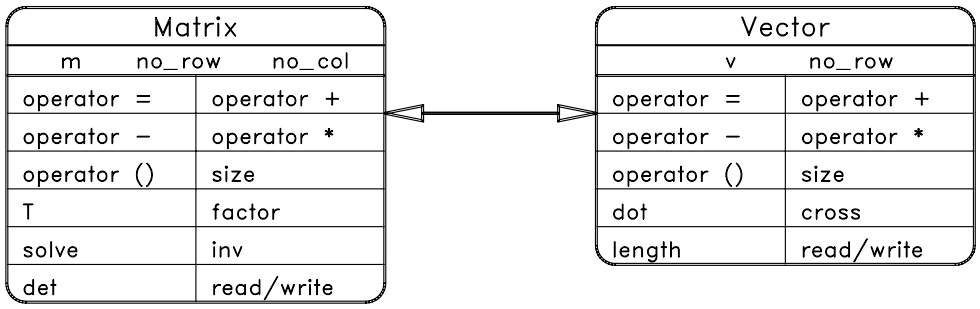


Figure 3.8: The Matrix and Vector classes

vector calculus

```
x = length(a)
y = dot(a,b)
c = cross(a,b)
```

where x, y are scalars and c is a Vector.

Solution methods are highly related to the matrix structure and they are defined in conjunction with the Matrix and Vector classes. These methods - being numerical algorithms - are programmed in procedural style taking the matrices and vectors as arguments. The solution of linear equation systems uses factorization, factor, and a series of substitutions, solve, to obtain the solution for a load pattern. The solution methods are also used by det to calculate the determinant and by inv to calculate the inverse of a square matrix. Thus the solution of a system of linear equations can be written directly,

inversion

```
a = inv(K) * f
```

or as a sequence of operations,

factor & solve

```
factor(K)
solve(K,a,f)
```

The general class definitions are given in Figure 3.8. Detailed description of the entire algebraic classes is given in Appendix A.

3.5 The application

This section presents an application that uses the defined classes. The application is valid for all types of linear problems with the exception of the model definition which requires knowledge of the specific types of elements or materials that are used in the analysis. The necessary modifications when introducing new elements are, however, restricted to a few additions in the command interpreter. The application will first be described by its tasks and then the full application will be summarized in pseudo-code in Algorithm 3.1.

3.5.1 Model definition

In this phase the model is defined. Usually, it is done taking input from the terminal, a file or a graphical environment. Thus, first part of the application is devoted to interpretation of input. A command interpreter reads the input and recognizes a number of keywords. Some keywords are shared by all models, e.g. *Node*, *Property*, *Vload*. Others are used to specify the specific elements and materials, e.g. *Solid3D8* which refers to a 8 node element for 3D solid mechanics or *Elastic* defining an isotropic elastic material. As an element in the following is referred to as Element, it is this phase that the variable is typecasted, meaning that the attributes and methods are initialized according to the element type. Due to the dynamic binding the correct implementations of the methods will be used in the following. The command interpreter must know which elements and materials are available, so introducing a new element requires the programmer to add a new keyword. However, being the only part of the application that distinguishes between different types of elements and materials, the command interpreter is the only part of the application that needs to be modified when introducing new elements or materials.

The command interpreter stores each model item in the linked lists, so that the rest of the application knows where to find its informations. A typical sequence may be the identification, typecasting, storage and calling a read method, e.g.

model definition

```
if (key = Solid3D8)
   elem = new Solid3D8
   ellist.add(elem)
   elem.read()
```

The new command symbols the creation of a new object and its initialization. The sequence given above may be defined for all types of model input, i.e. nodes, materials, properties and boundary conditions. These are collected in a command interpreter module, read_input, which in the end returns linked lists containing all items in the model.

3.5.2 Model generation

Having obtained the model input it is possible to start the model generation. This lies in the hands of 2 methods: Element.generate_model and Node.set_dofno. The first method uses the linked lists to establish connections between the elements and the connected nodes, material and property. All elements in the model requires the information so the application must loop over all elements in the element list. Having obtained information about the number of active dof the nodes are able to define the global dof numbers, thus the model generation consists of two loops: one over all elements and a loop over all nodes,

model generation

```
for (elem=ellist.start() to ellist.end) do
  elem.generate_model(nolist,matlist,prolist)

no_dof = 0
for (node=nolist.start() to nolist.end) do
  node.set_dofno(no_dof)
```

The total number of dof, no_dof, is initially set to 0. The set_dofno method assigns a unique dof number to all dof in the node and returns the updated number of dof, eventually giving no_dof. The loops run over all types of elements and nodes - due to the typecasting which is performed in the model definition phase each Item itself knows what version of the virtual methods to evoke.

3.5.3 Forming the global equation system

When the dimensions of the global matrices and vectors are known (= no_dof) they may be allocated and assembled. The assembly of the global equation system involves the prescribed values from the nodes, dof and load, and the processed element data, stiffness and load. Two loops are introduced for this purpose,

form equations

```
Matrix K(no_dof,no_dof)
Vector a(no_dof)
Vector f(no_dof)
IntArray fix(no_dof)

no_dof = 0
for (node=nolist.start() to nolist.end) do
  node.send_disp(a,fix)
  node.send_load(f)

for (elem=ellist.start() to ellist.end) do
  elem.assm_stiffness(K)
  elem.assm_load(f)
```

In C and C++ boolean variables are represented by integer variables: TRUE=1 and FALSE=0, thus the fix array may be defined as a 1D array of integers, cf. Appendix A.

3.5.4 Solving the global equation system

The global equation system is a constrained system of linear equations that can be solved explicitly. The solution process is divided in factorization, factor, of the global stiffness matrix and a solution procedure, solve, that uses the factorized matrix and the global dof and load vectors. These solution methods relate, as described previously, to the Matrix class that is used. Full square matrices uses **LU** factorization, while for symmetric matrices

the $\mathbf{LDL}^T$ scheme is available. A constrained system is characterized by having unknown values in both the dof and the load vector. The terms related to the prescribed dof are marked by the fix array and are omitted in the factorization. The unknown dof and reactions are then determined by the solve method. An $\mathbf{LDL}^T$ scheme for profile matrices has been developed, Hededal & Krenk (1993), where the unknown dof and reactions are found simultaneously during the substitution process. The algorithm is included in the C++ ProMatrix class but to retain the numerical efficiency it is implemented using a pure C syntax, see Appendix A.

solve equations

```
factor(K,fix)
solve(K,a,f,fix)

for (node=nolist.start() to nolist.end) do
   node.get_disp(a)
   node.get_load(f)
```

After the analysis the Nodes retrieve the unknown dof and loads. It is thereby possible to free the memory used by the global system and use it for other purposes during the postprocessing phase.

3.5.5 Postprocess

Last part of the application is to output the various results from the analysis. The dof and reactions do not require processing and may be given directly in tables. The element results, strain and stress, are derived from the dof which the Element get from the Nodes and presented in tables.

postprocess

```
for (node=nolist.start() to nolist.end) do
   node.write_disp

for (node=nolist.start() to nolist.end) do
   node.write_load

for (elem=ellist.start() to ellist.end) do
   elem.write_strain

for (elem=ellist.start() to ellist.end) do
   elem.write_stress
```

The virtual methods, strain and stress, are called by the 2 write methods. Again, it should be noted that the elements in the List are not necessarily of the same type.

3.5.6 Linear and non-linear applications

The segments described above form an application for linear analysis which is summarized in Algorithm 3.1. The input and output parts, which are hidden in read_input and write_output, may vary from problem to problem, whereas the other parts apply for all sorts of linear analyses.

Extension from linear to non-linear analysis mainly affects the solution part of the application. The load is applied in a number of load steps and iterations are performed to reestablish equilibrium. The extension thus consists of adding 2 control structures: a load incrementation and equilibrium iterations. A modified Newton-Raphson scheme may have the following form

Newton-Raphson

```
Solve global equation system:
  for (n=1 to no_loadstep) do
    for (elem=ellist.start() to ellist.end) do
      elem.assm_stiffness(K)
    factor(K,fix)
    fe += df
    solve(K,da,df,fix)
    do
      for (elem=ellist.start() to ellist.end) do
        elem.assm_intforce(fi,a+da)
      r = fe – fi
      solve(K,delta,r,fix)
      da += delta
    until (norm(r) < EPS*norm(fe))
    a += da
    for (node=nolist.start() to nolist.end) do
      node.get_dof(a)
      node.get_load(fe)
```

In each load step or even in iteration a representative stiffness must be evaluated, thus the assembly of the stiffness matrix is moved into these loops. The solution algorithm is driven by the unbalance between the internal forces, fi, corresponding to the current dof and the external loads, fe, thus in order to obtain a new increment, delta, the residual, r = fe – fi, must be evaluated in each iteration.

In the more general methods both the load and the dof increments are adjusted during the iterations, Crisfield (1981), Krenk (1993b), Krenk & Hededal (1993). This is necessary if the equilibrium path has limit points or snap-through and may as well improve the convergence rate. Implementation of such methods require very few additions to the Newton-Raphson scheme above, as demonstrated in Chapter 6. Furthermore, it is notable that these algorithms are implemented on the application level, thus do not require the programmer to modify the rest of the system.

ALGORITHM 3.1: LINEAR ANALYSIS

```
Variables:
  elem, node
  ellist, nolist, matlist, prolist
  no_dof, K, a, f, fix

Model definition:
  read_input(all lists)

Model generation:
  for (elem=ellist.start() to ellist.end) do
    elem.generate_model(nolist,matlist,prolist)
  no_dof = 0
  for (node=nolist.start() to nolist.end) do
    node.set_dofno(no_dof)

Form global equation system:
  allocate K, a, f, fix
  for (node=nolist.start() to nolist.end) do
    node.send_disp(a,fix)
    node.send_load(f)
  for (elem=ellist.start() to ellist.end) do
    elem.assm_stiffness(K)
    elem.send_load(f)

Solve global equation system:
  factor(K,fix)
  solve(K,a,f,fix)
  for (node=nolist.start() to nolist.end) do
    node.get_disp(a)
    node.get_load(f)

Postprocess:
  write_output(nolist,ellist)
```

Chapter 4

Customizing the FEM classes

The FEM classes defined in the previous section formulate a framework for implementing elements and materials. In this chapter is demonstrated how the FEM classes can be customized to finite elements for linear potential problems. The aim is to introduce a 4 point standard approach that can be used for specializing the FEM classes.

1. Theory and finite element formulation
2. Identification of parameters and virtual methods, cf. Table 2.4
3. Additional attributes and methods
4. Additional classes

For the potential problem the theory and finite element formulation are established in Section 2.1 and Section 2.2. This chapter starts by giving an interpretation of the virtual methods. The isoparametric element concept will be used, leading to the specific formulation of element matrices and vectors and a numerical integration scheme that uses Gauss quadrature. A Continuum element class formulated as isoparametric elements is presented and a Gausspoint class introduced. The Continuum element constitutes a class of elements that in this chapter is specialized to 2D and 3D potential elements and in the following chapter extended to include solid elements.

4.1 Potential element with linear materials

From the formulation given in Section 2.2 it is found that a potential element with n nodes (no_nodes=n) is described by 1 dof pr. node (no_dof=1), i.e.

$$\mathbf{a}^T = [\, a^1 \quad a^2 \quad \cdots \quad a^n \,] \tag{4.1}$$

$$\mathbf{f}^T = [\, f^1 \quad f^2 \quad \cdots \quad f^n \,] \tag{4.2}$$

giving a simple form of the shape function matrix,

$$\mathbf{N} = [\, N^1 \quad N^2 \quad \cdots \quad N^n \,] \tag{4.3}$$

The strain, $\boldsymbol{\varepsilon}$, which is this case is the gradient of the potential, $\boldsymbol{\nabla} u$, has DIM components (no_strain=DIM), where DIM is the spatial dimension. The strain may be derived from the

discretized dof, $\mathbf{a}$, using the gradient matrix, $\mathbf{B}$,

$$\boldsymbol{\varepsilon} = \begin{bmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \varepsilon_3 \end{bmatrix} = \mathbf{B}\,\mathbf{a} \tag{4.4}$$

where the gradient matrix, $\mathbf{B}$, is a DIM$\times$no_nodes matrix, i.e.

$$\mathbf{B} = \begin{bmatrix} \partial N^1/\partial x_1 & \partial N^2/\partial x_1 & \cdots & \partial N^n/\partial x_1 \\ \partial N^1/\partial x_2 & \partial N^2/\partial x_2 & \cdots & \partial N^n/\partial x_2 \\ \partial N^1/\partial x_3 & \partial N^2/\partial x_3 & \cdots & \partial N^n/\partial x_3 \end{bmatrix} \tag{4.5}$$

In potential problems the flux becomes the generalized stress, $\boldsymbol{\sigma}$. It is found from the strain using the constitutive relation,

$$\boldsymbol{\sigma} = \begin{bmatrix} \sigma_1 \\ \sigma_2 \\ \sigma_3 \end{bmatrix} = \mathbf{C}\,\boldsymbol{\varepsilon} \tag{4.6}$$

For linear isotropic materials $\mathbf{C}$ has the form

$$\mathbf{C} = c\,\mathbf{I} \tag{4.7}$$

where c is the material constant (Material.no_par=1) and $\mathbf{I}$ is the unit matrix with dimensions equal to the spatial dimension. More general anisotropic materials requires a full constitutive matrix on the form

$$\mathbf{C} = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix} \tag{4.8}$$

where c_{ij} are material constants. Assuming, however, that the constitutive relation can be given on potential form imposes symmetry on the constitutive matrix thus reducing the number of independent parameters from 9 to 6 (Material.no_par=6). This assumption is used in ObjectFEM.

Having defined the constitutive matrix leads to calculation of the element stiffness matrix,

$$\mathbf{K}^e = \int_\Omega \mathbf{B}^T\,\mathbf{C}\,\mathbf{B}\,d\Omega \tag{4.9}$$

which is an $n \times n$ symmetric matrix. Calculation of the volume loads, $\mathbf{f}_v^e$, is dependent on the load field, $Q(\mathbf{x})$. The field may be approximated from the node intensities, $\mathbf{Q}^T = [\,Q^1 \quad Q^2 \quad \cdots \quad Q^n\,]$, using the shape functions, $\mathbf{N}(\mathbf{x})$,

$$Q(\mathbf{x}) = \mathbf{N}(\mathbf{x})\,\mathbf{Q} \tag{4.10}$$

whereby the integral over the element becomes

$$\mathbf{f}_v^e = \int_\Omega \mathbf{N}^T \, \mathbf{N} \, d\Omega \; \mathbf{Q} \tag{4.11}$$

For a surface, i, the surface load field, $q_n^i(\mathbf{x})$, may be interpolated from the node intensities, $\mathbf{q}_n^i = [\, q_n^{i,1} \quad q_n^{i,2} \quad \cdots \quad q_n^{i,n} \,]$, by use of the shape functions, i.e.

$$q_n^i(\mathbf{x}) = \mathbf{N}(\mathbf{x}) \, \mathbf{q}_n^i \tag{4.12}$$

The integral of the surface load, $\mathbf{f}_s^e$, consists of a sum of s surface integrals, s being the number of element surfaces. In principle the integration is

$$\mathbf{f}_e^s = \sum_{i=1}^{s} \left[\int_{\Gamma_i} \mathbf{N}^T \, \mathbf{N} \, d\Gamma \right] \; \mathbf{q}_n^i \tag{4.13}$$

This form is in practice expensive because most of the entries in the matrices are 0. Instead a more compact form may be chosen where only the non-zero parts are used.

Defining the number of nodes in each direction and the shape functions conclude the element formulation. In ObjectFEM the number of nodes is the same for all directions.

4.2 Isoparametric elements

A finite element solution is an approximation to the exact one. Using infinitely many and infinitely small elements to discretize the domain the solution should converge to the exact solution. This convergence requirement can be fulfilled if the element shape functions are complete and compatible. Completeness is stated as the ability to represent constant dof and strain fields within the element. Compatibility means that the field, represented by the shape functions and the dof, is continuous over element boundaries, Ottosen & Petersson (1992). For arbitrary geometries compatibility is difficult to obtain if the mesh is not aligned with coordinate axis. However, using the isoparametric element concept the mesh can be mapped onto a standard domain where it is aligned with the basis coordinate axis, thus making it possible to establish compatibility for domains with curved boundaries. In this section the isoparametric element formulation, which is later used for potential elements and solid elements, is established.

A basis element is described in the basis coordinates, $\boldsymbol{\xi}$. Each point, $\boldsymbol{\xi}$, in the basis element corresponds to a point, $\mathbf{x}$, in the physical domain, see Figure 4.1. The mapping between the 2 coordinate systems is described by the shape functions, $\mathbf{N}(\boldsymbol{\xi})$, and the node coordinates in the physical domain. Each component x_i is described by a nodal component vector, $\mathbf{X}_i^T = [\, X_i^1 \quad X_i^2 \quad \cdots \quad X_i^n \,]$,

$$x_i(\boldsymbol{\xi}) = \mathbf{N}(\boldsymbol{\xi}) \, \mathbf{X}_i \tag{4.14}$$

Notice that the shape functions are expressed in the basis coordinates.

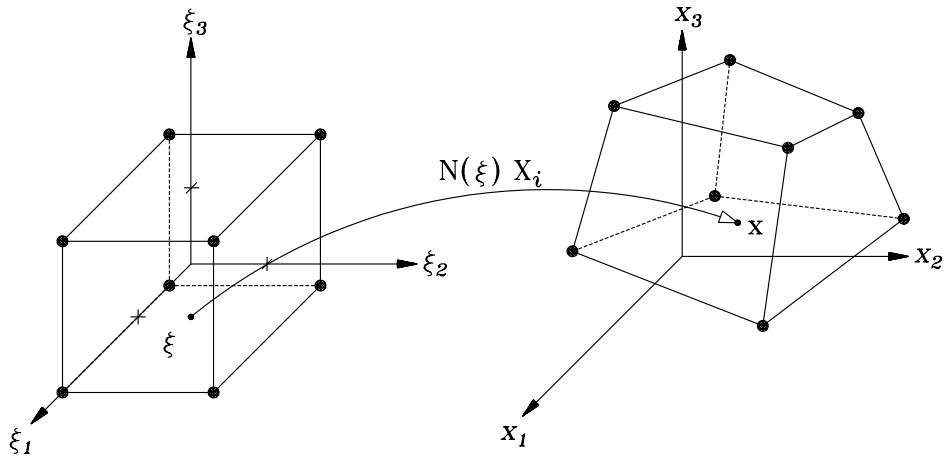


Figure 4.1: Isoparametric element

The dof, $u(\mathbf{x})$, is similarly expressed from the nodal values, $\mathbf{a}$, by the relation

$$u(\mathbf{x}(\boldsymbol{\xi})) = \mathbf{N}(\boldsymbol{\xi})\,\mathbf{a} \tag{4.15}$$

The strain components, ε_i, are here the gradient components found by partial differentiations,

$$\varepsilon_i = \frac{\partial u}{\partial x_i} = \sum_j \frac{\partial u}{\partial \xi_j}\frac{\partial \xi_j}{\partial x_i} \quad ; \quad i = 1,2,3 \tag{4.16}$$

Inserting the mapping, (4.15), and introducing matrix notation yields

$$\begin{bmatrix} \varepsilon_1 & \varepsilon_2 & \varepsilon_3 \end{bmatrix} = \begin{bmatrix} \dfrac{\partial \mathbf{N}(\boldsymbol{\xi})}{\partial \xi_1}\mathbf{a} & \dfrac{\partial \mathbf{N}(\boldsymbol{\xi})}{\partial \xi_2}\mathbf{a} & \dfrac{\partial \mathbf{N}(\boldsymbol{\xi})}{\partial \xi_3}\mathbf{a} \end{bmatrix} \begin{bmatrix} \dfrac{\partial \xi_1}{\partial x_1} & \dfrac{\partial \xi_1}{\partial x_2} & \dfrac{\partial \xi_1}{\partial x_3} \\ \dfrac{\partial \xi_2}{\partial x_1} & \dfrac{\partial \xi_2}{\partial x_2} & \dfrac{\partial \xi_2}{\partial x_3} \\ \dfrac{\partial \xi_3}{\partial x_1} & \dfrac{\partial \xi_3}{\partial x_2} & \dfrac{\partial \xi_3}{\partial x_3} \end{bmatrix} \tag{4.17}$$

Finally, transposing (4.17) the strain is calculated as

$$\boldsymbol{\varepsilon} = \mathbf{J}^{-1}(\boldsymbol{\xi})\,\boldsymbol{\nabla}_\xi \mathbf{N}(\boldsymbol{\xi})\,\mathbf{a} \tag{4.18}$$

where $\boldsymbol{\nabla}_\xi$ is the gradient with respect to the basis coordinates that for $\mathsf{DIM} = 3$ is given as

$$\boldsymbol{\nabla}_\xi = \begin{bmatrix} \dfrac{\partial}{\partial \xi_1} \\ \dfrac{\partial}{\partial \xi_2} \\ \dfrac{\partial}{\partial \xi_3} \end{bmatrix} \tag{4.19}$$

The relation (4.18) identifies the gradient matrix, $\mathbf{B}(\mathbf{x})$, as the matrix product of a gradient field, $\boldsymbol{\nabla}_\xi \mathbf{N}(\boldsymbol{\xi})$, and a transformation matrix, $\mathbf{J}(\boldsymbol{\xi})$,

$$\mathbf{B}(\mathbf{x}) = \mathbf{J}^{-1}(\boldsymbol{\xi})\, \boldsymbol{\nabla}_\xi \mathbf{N}(\boldsymbol{\xi}) \tag{4.20}$$

$\mathbf{J}$ is the Jacobi matrix which expresses the differential mapping between 2 domains. It is defined as

$$\mathbf{J} = \begin{bmatrix} \dfrac{\partial x_1}{\partial \xi_1} & \dfrac{\partial x_2}{\partial \xi_1} & \dfrac{\partial x_3}{\partial \xi_1} \\ \dfrac{\partial x_1}{\partial \xi_2} & \dfrac{\partial x_2}{\partial \xi_2} & \dfrac{\partial x_3}{\partial \xi_2} \\ \dfrac{\partial x_1}{\partial \xi_3} & \dfrac{\partial x_2}{\partial \xi_3} & \dfrac{\partial x_3}{\partial \xi_3} \end{bmatrix} \tag{4.21}$$

The Jacobi matrix may be evaluated using the mapping, (4.14),

$$\frac{\partial x_i}{\partial \xi_j} = \frac{\partial \mathbf{N}(\boldsymbol{\xi})}{\partial \xi_j}\, \mathbf{X}_i \quad ; \quad i,j = 1,2,3 \tag{4.22}$$

On matrix this becomes

$$\mathbf{J} = \boldsymbol{\nabla}_\xi \mathbf{N}(\boldsymbol{\xi})\, \mathbf{X} \tag{4.23}$$

where the $\mathbf{X}$ matrix is formed by the 3 coordinate vectors, i.e.

$$\mathbf{X} = [\, \mathbf{X}_1 \quad \mathbf{X}_2 \quad \mathbf{X}_3 \,] \tag{4.24}$$

It is notable that the gradient of the shape functions, $\boldsymbol{\nabla}_\xi \mathbf{N}(\boldsymbol{\xi})$, is used both for calculating the Jacobi matrix, $\mathbf{J}(\boldsymbol{\xi})$, and setting up the gradient matrix, $\mathbf{B}(\mathbf{x})$. These task should be collected so that $\boldsymbol{\nabla}_\xi \mathbf{N}(\boldsymbol{\xi})$ is evaluated only once for each point.

A fundamental characteristic for isoparametric elements is that the integrals can be evaluated in the basis domain, $\xi_i \in [-1;1]$. Substitution from the physical domain to the basis element is done by introducing the determinant of the Jacobi matrix, $J = |\mathbf{J}|$,

$$d\Omega = dx_1\, dx_2\, dx_3 = J\; d\xi_1\, d\xi_2\, d\xi_3 \tag{4.25}$$

Thus the Jacobian, J, expresses the ratio between a unit volume in the 2 domains. The integrations can be performed in the basis coordinates replacing the volume integrals by triple integrals, i.e.

$$\mathbf{K}^e = \int_{-1}^{1}\int_{-1}^{1}\int_{-1}^{1} \mathbf{B}^T(\boldsymbol{\xi})\, \mathbf{C}\, \mathbf{B}(\boldsymbol{\xi})\, J(\boldsymbol{\xi})\; d\xi_1 d\xi_2 d\xi_3 \tag{4.26}$$

$$\mathbf{f}_v^e = \left[\int_{-1}^{1}\int_{-1}^{1}\int_{-1}^{1} \mathbf{N}^T(\boldsymbol{\xi})\, \mathbf{N}(\boldsymbol{\xi})\, J(\boldsymbol{\xi})\; d\xi_1 d\xi_2 d\xi_3\right] \mathbf{Q} \tag{4.27}$$

The surface integral is replaced by a sum over s double integrals

$$\mathbf{f}_s^e = \sum_{i=1}^{s} \left[\int_{-1}^{1}\int_{-1}^{1} \mathbf{N}^T(\boldsymbol{\xi})\, \mathbf{N}(\boldsymbol{\xi})\, J_S(\boldsymbol{\xi})\; d\xi_1 d\xi_2\right] \mathbf{q}_n^i \quad ; \quad \xi_3^i = \pm 1 \tag{4.28}$$

where ξ_3^i refers to the coordinate which is constant for surface i. J_S is the surface version of the Jacobian giving the ratio between a unit area in a physical domain and the corresponding area in the basis domain.

4.2.1 Numerical integration

Numerical integration schemes replaces the integrals with summations. The value of the function, $f(x)$, is evaluated in a finite number of point and a weighted sum is taken, e.g.

$$\int f(x)\,dx = \sum_{i=1}^{g} f(x^i)\,w^i \tag{4.29}$$

For polynomials the Gauss quadrature is efficient, Ottosen & Petersson (1992): nth order Gauss perform exact integrations of polynomials of order $2n-1$. The finite element shape functions are usually low order polynomials, thus Gauss quadrature is a natural choice for integrating the element integrals. Furthermore, being defined for standard domains the isoparametric elements have explicitly determined Gauss coordinates and weights, which can be given as tables for each element.

The integration is carried out in one direction at the time, i.e. there are n Gauss points in each direction, thus for 3D problems n^3 points are needed. The position of the Gauss points are the same in all directions and are given as a coordinate set, $\boldsymbol{\xi}^i$. For volume integrals the triple integrals are replaced by a single weighted sum,

$$\begin{aligned} \mathbf{K}^e &= \sum_{i=1}^{g} \left[\mathbf{B}^T(\boldsymbol{\xi}^i)\,\mathbf{C}\,\mathbf{B}(\boldsymbol{\xi}^i)\,\Delta\Omega^i\right] \\ \mathbf{f}_v^e &= \sum_{i=1}^{g} \left[\mathbf{N}^T(\boldsymbol{\xi}^i)\,\mathbf{N}(\boldsymbol{\xi}^i)\,\Delta\Omega^i\right]\,\mathbf{Q} \end{aligned} \tag{4.30}$$

where the volume weight, $\Delta\Omega^i$, is defined as

$$\Delta\Omega^i = J(\boldsymbol{\xi}^i) \prod_{j=1}^{DIM} w_j^i \tag{4.31}$$

Thus the resulting weight is found as the product of the individual weights for each of the spatial dimensions. Similarly, it is possible to replace the double integrals in the surface load with simple sums, i.e.

$$\mathbf{f}_s^e = \sum_{l=1}^{s} \left[\sum_{i=1}^{g_S} \left[\mathbf{N}^T(\boldsymbol{\xi}^i)\,\mathbf{N}(\boldsymbol{\xi}^i)\,\Delta\Gamma^i\right]\right]\,\mathbf{q}_n^l \quad ; \quad \xi_k^i = \pm 1 \tag{4.32}$$

The surface weight, $\Delta\Gamma^i$, is evaluated as

$$\Delta\Gamma^i = J_S(\boldsymbol{\xi}^i) \prod_{j=1, j\neq k}^{DIM} w_j^i \tag{4.33}$$

As the polynomial degree is usually the same for volume and surface integrals the same Gauss points and weights can be used with the exception that the coordinate that is constant for surface l is set $\xi_k^i = \pm 1$ and the weight is $w_k^i = 1$. Also the number of integration points should be reduced 1 order, e.g. from n^3 to n^2.

An element is thus apart from the number of nodes and the shape functions defined by the order of the Gauss integration scheme for each direction. In ObjectFEM the Gauss order is the same for all directions and set as the total number of points, e.g. no_gauss $= n^3$.

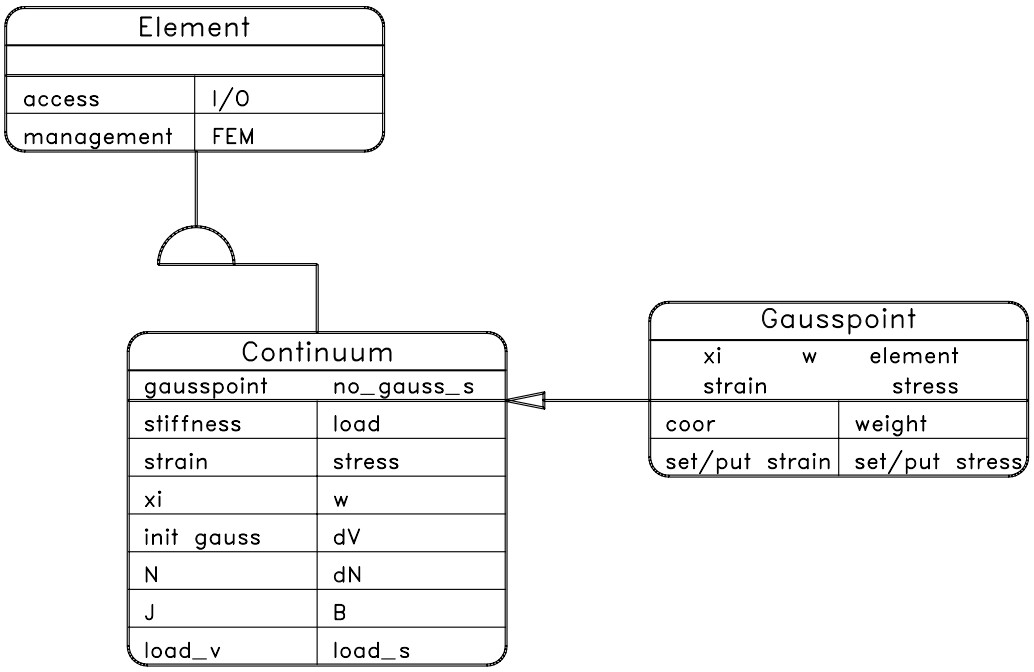


Figure 4.2: Isoparametric elements and Gausspoint

4.3 The isoparametric element class

The previous sections give a common definition of the isoparametric element. It may be considered as an interpretation of the virtual methods in terms of a set of new methods.

The preprocesssing tasks of an isoparametric element is to evaluate the volume and surface integrals. For that purpose it should define implementations of the virtual methods, stiffness and load. The integrations are carried out by Gauss quadrature. A Gausspoint class is introduced to manage the coordinates and weights of the single Gauss point. The isoparametric element class adds an array called gausspoint to the attribute list, which stores the no_gauss Gausspoint. Each Gausspoint is initialized with its basis coordinates and weights and a reference to the Element by the method init_gauss:

init gauss

```
Continuum.init_gauss():
   gx = xi()
   gw = w()
   for (i=1 to no_gauss) do
      gausspoint(i) = new Gausspoint(this,gx(i),gw(i))
```

The method is static, but uses two virtual methods, xi and w, that set up matrices with all the Gauss coordinates and weights. Systematic generation of the Gauss points and weights can be obtained by looking at one direction at the time. The Gauss coordinate and weight form a pair, (ξ, w), which can be combined with other pairs to form the 2 and 3 dimensional Gauss schemes. Introducing a matrix, Xi, which contains the node coordinates in the basis element gives that a 2nd order Gauss scheme may be initialized the following

way:

xi

```
Continuum.xi():
   gx = 1/sqrt(3)
   return = Xi * gx
```

A 4 node 2D element has the node coordinates, Ξ, thus the coordinates for 2nd order Gauss becomes

$$\Xi = \begin{bmatrix} -1 & -1 \\ 1 & -1 \\ 1 & 1 \\ -1 & 1 \end{bmatrix} \quad \Rightarrow \quad \boldsymbol{\xi} = \begin{bmatrix} -1/\sqrt{3} & -1/\sqrt{3} \\ 1/\sqrt{3} & -1/\sqrt{3} \\ 1/\sqrt{3} & 1/\sqrt{3} \\ -1/\sqrt{3} & 1/\sqrt{3} \end{bmatrix}$$

For 2nd order Gauss the weights are all 1, thus

w

```
Continuum.w():
   Matrix gw(DIM,no_nodes)
   gw = 1
   return gw
```

where the assignment, gw=1, sets all components in the matrix equal to 1. The simple format is due to very special form of 2nd order Gauss. 3rd order Gauss uses 3 points in each direction and a similar scheme may be employed defining the center node as number 9, see e.g. p. 538 in Zienkiewicz & Taylor (1989).

This enables evaluation of the integrals by looping over all the no_gauss points. The stiffness thus becomes

stiffness

```
Continuum.stiffness():
   c = material.C()
   for (i=1 to no_gauss) do
      dn = dN(i)
      jacobi = J(dn)
      b = B(dn,jacobi)
      dv = dV(i,jacobi)
      Ke += b.T * c * b * dv
   return Ke
```

where the material is requested to supply the constitutive matrix, C. To evaluate the stiffness contribution for each Gauss point 4 additional methods have been introduced: dN, J, B and dV.

The method dN evaluates and sets up the gradient, $\boldsymbol{\nabla}_\xi \mathbf{N}$, for gausspoint(g). The shape functions may be generated as a product of 1 dimensional functions, $N_j^i(\xi_j)$, in the following way:

$$N^i(\boldsymbol{\xi}) = \prod_{j=1}^{DIM} N_j^i(\xi_j) \tag{4.34}$$

The notation, N_j^i, refers to the shape function in direction j for node number i and ξ_j is the value of the basis coordinate in direction j. A component in the gradient matrix, $\boldsymbol{\nabla}_\xi \mathbf{N}$, thereby becomes

$$\frac{\partial N^i(\boldsymbol{\xi})}{\partial \xi_j} = \frac{\partial N_j^i(\xi_j)}{\partial \xi_j} \prod_{k=1, k\neq j}^{DIM} N_j^i(\xi_k) \tag{4.35}$$

Translating this into pseudo-code gives

dN

```
Continuum.dN(g):
  Matrix dn(DIM,no_nodes)
  x = gausspoint(g).coor()
  for (i=1 to no_nodes) do
    for (j=1 to DIM) do
      dn(j,i) = dshape(Xi(i,j),x(i,j))
      for (k=1 to DIM) do
        if (k ≠ j)
          dn(j,i) *= shape(Xi(i,k),x(i,k))
  return dn
```

The 2 methods, shape and dshape, evaluate the value of the 1 dimensional shape function and its derivative on basis of the node coordinate, Xi, and the Gauss coordinate, x. The actual implementation in ObjectFEM differs from the presentation given here, thus these methods are not included in the class description in Figure 4.2.

This matrix is used to calculate the Jacobi matrix, $\mathbf{J}$. The method J uses the matrix, X, which is set up from the nodal coordinates in the physical domain.

Jacobi

```
Continuum.J(dn):
  for (i=1 to no_nodes) do
    for (j = 1 to DIM) do
      X(i,j) = nodes(i).put_coor(j)
  return dn*X
```

The B method uses those 2 matrices to set up the gradient matrix, $\mathbf{B}(\mathbf{x})$. For a potential problem this simply consists of a transformation:

B

```
Continuum.B(dn,J):
  return inv(J) * dn
```

where inv performs a numerical inversion of the Jacobi matrix.

The volume weight dV is also defined as a virtual method, whereby the integration of the stiffness becomes identical for 3D as well as 2D problems, cf. (4.31). In 3D it is the product of the weights and the Jacobian, J.

dV - 3D

```
Continuum.dV(g,J):
   return gausspoint(g).weight() * det(J)
```

The method is virtual and can be overwritten so that for 2D problems is becomes

dV - 2D

```
Conti2D.dV(g,J):
   return Continuum.dV(g,J) * property.put_par(1)
```

where the thickness is obtained as parameter No. 1 from the property. The call with prefix, Continuum.dV, explicitly tells the program to use the implementation in the superclass. If the prefix is not used the call would be recursive and result in an interminable loop.

The load vector is a sum of the volume loads and surface load, i.e.

load

```
Continuum.load():
   return load_v()+load_s()
```

It receives contribution from two 2 methods: load_v and load_s. The volume load is essentially the same as the integration of the stiffness matrix, i.e.

load v

```
Continuum.load_v():
   for (i=1 to no_gauss) do
      dn = dN(i)
      jacobi = J(dn)
      dv = dV(i,jacobi)
      n = N(i)
      F += n.T * n * dv
   return F * vload
```

The shape functions in a Gauss point can be evaluated from (4.34). The shape function matrix is set up by a virtual method, N, defined as

N

```
Continuum.N(g):
   Matrix n(1,8)
   x = gausspoint(g).coor()
   for (i=1 to no_nodes) do
      n(1,i) = 1
      for (j=1 to DIM) do
         n(1,i) *= shape(Xi(i,j),x(i,j))
   return n
```

The volume load is also identical for 2D and 3D and may therefore be defined statically.

The form of the surface load given in (4.12) is not very convenient to program. There are several ways to do it which is closely related to the type of surface loads. Special type

of loadings like those that act normal to the surface, e.g. normal flux or pressure, should be handled one way while more general loading types require other means. Another issue that is hard to generalize is the numbering sequence of the surface nodes. There must be a specification of which nodes belong to which surface, this is e.g. dependent on whether the element has 8 or 20 nodes. Therefore the method load_s will be defined as a virtual method. The integration is performed with no_gauss_s Gauss points. In ObjectFEM this corresponds to the same order as the volume integration scheme and the coordinates and weights may be obtained from the Gausspoints , see e.g. Section 4.2.1.

The postprocessing part consists of evaluating the strain and stress from the node dof. The isoparametric element must supply an appropriate definitions of the 2 virtual methods, strain and stress. For this purpose the gradient matrix, $\mathbf{B}(\mathbf{x})$, must be available - either it could be stored during preprocessing or it is reevaluated at this point. The strain may be evaluated as

strain

```
Continuum.strain():
   a = get_dof()
   for (i = 1 to no_gauss) do
      dn = dN(i)
      jacobi = J(dn)
      gausspoint(i).set_strain() = B(dn,jacobi) * a
```

The stress is evaluated from the strain by multiplying with the constitutive matrix.

stress

```
Continuum.stress():
   if (gausspoint(1).put_strain = 0)
      strain()
   c = material.C()
   for (i = 1 to no_gauss) do
      gausspoint(i).set_stress() = c * gausspoint(i).put_strain()
```

The `if` statement tests whether the strain has already been evaluated - if not strain is call before proceeding. Both methods, strain and stress, are the same for all isoparametric elements with linear materials.

The strain and stress are field properties, but it is only convenient to evaluate them at selected points. The Gauss points are characteristic by being those points in the element for which the convergence rate for strain is the highest. The strain and stress calculated in these points are thus the best possible approximations with a given discretization. Therefore the strain and stress are evaluated in the Gauss points and the Gausspoint class have means for obtaining and storing the strain and stress. In linear analysis this is not necessary because the strain and stress can be derived explicitly from the strain gradient and the node dof. It is, however, chosen to use the Gauss point because the elements thereby apply equally well to problems with linear as well as non-linear material models where the strain and stress are found from incremental relations, cf. Chapter 8.

4.3.1 The Gausspoint class

The Gausspoint class, presented in Figure 4.2, is introduced to manage the coordinates and weights for the numerical integration. The Gauss points are optimal when evaluating the strain and stress from the nodal dof. The Gausspoint class is able to store the strain and stress in the point and has methods for updating these corresponding to the current dof state. The importance of this will become more obvious for material non-linear problems such as plasticity theory.

The class has 3 attributes related to the task of defining the Gauss coordinates: element, xi, w. The element attribute contains a reference to the element that the point is part of. The xi and w are the coordinates and weights. These attributes are initialized upon creation, i.e. they are set by the Continuum.init_gauss method, which calls the constructor, viz.

Gausspoint

```
Gausspoint.constructor(elem,gx,gw)
  element = elem
  xi = gx
  w = gw
```

The coordinates and weights are presented to the element by 2 methods: coor returns a vector with the basis coordinates and weight returns the product of the weights. The default argument, s=0, identifies that the full product should be formed corresponding to a volume integration, (4.31). If s>0 the component s constant in the surface integrations and the weight should be omitted in the product, (4.33).

weight

```
Gausspoint.weight(s=0)
  gw = 1
  for (i=1 to w.size()) do
    if (i≠s)
      gw *= w(i)
  return gw
```

Using Vector.size makes the method identical for all dimensions. The C abbreviation gw *= w(i) is used to replace gw = gw * w(i).

The strain and stress describe the current state of the Gausspoint. The attributes may be set by the element using set_strain or set_stress and upon request returned by put_stress and put_stress, e.g.

set strain

```
Gausspoint.set_strain():
  return strain
```

where the attribute is set by assignment.

4.3.2 Potential elements in 2D and 3D

The Continuum class may be superclass for many different elements. One of the simplest is a 3D element with 8 nodes and linear shape functions for potential problems, Pot3D8. Pot3D8 will be used as superclass for all continuum elements. It delivers the implementation of init_gauss, J, stiffness, load_v, strain and stress which apply to all problems both in 2D and 3D. The superclass further defines a number of virtual methods: xi, w, dV, N, dN, B and load_s. These may be overwritten by the subclasses. ObjectFEM contains presently 4 isoparametric potential elements, which are all of the serendipity type, see e.g. Zienkiewicz & Taylor (1989). The 3D elements are Pot3D8 with linear shape functions and Pot3D20 which has quadratic shape functions. Pot2D4 and Pot2D8 are the linear and quadratic 2D elements, respectively. The hierarchy is presented in Figure 4.3.

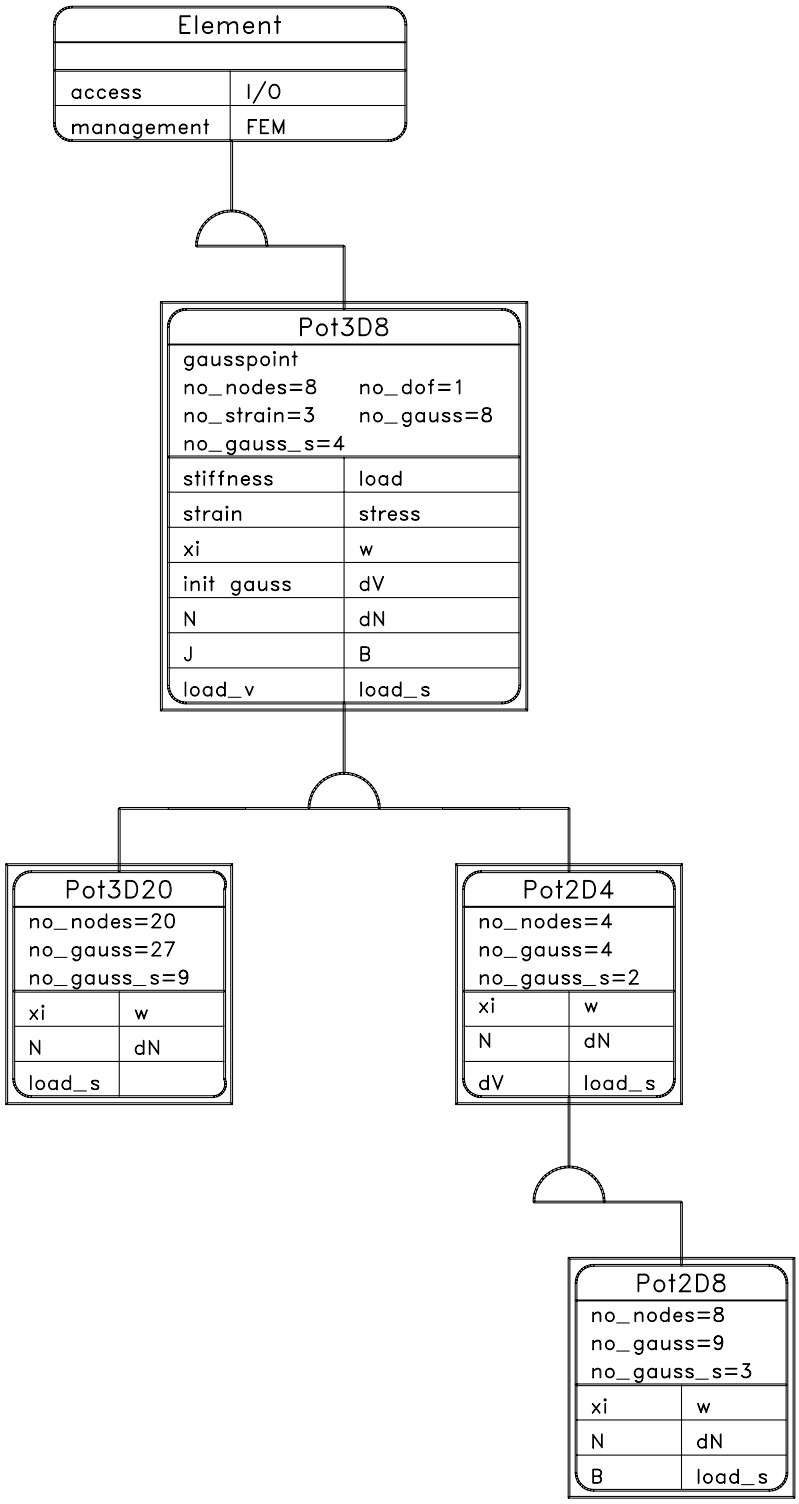


FIGURE 4.3: ISOPARAMETRIC POTENTIAL ELEMENTS

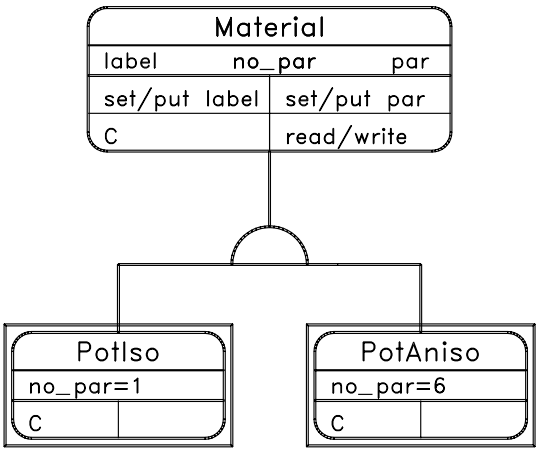


Figure 4.4: Specializing Material to potential problems

4.4 Customizing Material and Property

Two types of material models are defined for the potential problems: a simple linear isotropic material, (4.7), and an anisotropic material as defined by (4.8). For the isotropic linear material 2 simple material subclasses are defined: PotIso3D and PotIso2D which are described in terms of 1 parameter, c , i.e. no_par = 1. The constitutive matrix is easily set up as a diagonal matrix containing the single parameter.

C

```
PotIso3D.C():
   Matrix c(3,3)
   for (i to 3) do
      c(i,i) = par(1)
   return c
```

It could be part of the definition that the Element supplied information about the spatial dimension so that it is possible to use only one class. The full anisotropic material classes, PotAniso, are described by no_par = 6, and the constitutive matrix is set up as for PotIso. The Figure 4.4 presents the Material subclasses.

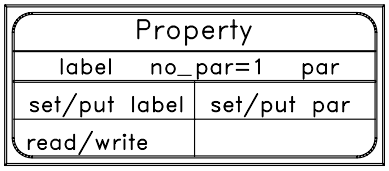


Figure 4.5: Specializing Property to plane problems

For plane problem it is necessary to set the thickness of the element. This is done by a Property object, see Figure 4.5. There is not defined a specific subclass, because it simply consists of storing and returning prescribed parameters. Initializing the object thus requires the user to set the number of parameters, no_par, explicitly, before it is possible

to store any values.

Chapter 5

Solid elements

This chapter describes how the isoparametric solid elements can be derived from the potential elements. The approach will follow the one outlined in the beginning of the previous section. Making use of the theoretical formulation given in Section 2.3 the problem parameters and concepts are resumed. Then the element classes are described with relation to the isoparametric potential elements concentrating on the methods that must be redefined to handle the extended number of dof and strain components. An linear elastic, isotropic material model is introduced for 3D problem and in conjunction with the 2D elasticity formulation the 3D class, Elastic, is specialized into classes for plane stress and plane strain. A hierarchy of isoparametric elements and linear elastic material models are summarized in Figure 5.1 and Figure 5.2.

5.1 Solid element for linear elasticity

From the formulation given in Section 2.3 it is found that a 3D solid element with n nodes (no_nodes=n) is described by 3 dof pr. node (no_dof=3), i.e.

$$\mathbf{a}^T = [\, a_1^1 \quad a_2^1 \quad a_3^1 \quad \cdots \quad a_1^n \quad a_2^n \quad a_3^n \,] \tag{5.1}$$

$$\mathbf{f}^T = [\, f_1^1 \quad f_2^1 \quad f_3^1 \quad \cdots \quad f_1^n \quad f_2^n \quad f_3^n \,] \tag{5.2}$$

giving the shape function matrix, $\mathbf{N}$, on expanded form,

$$\mathbf{N} = \begin{bmatrix} N^1 & 0 & 0 & \cdots & N^n & 0 & 0 \\ 0 & N^1 & 0 & \cdots & 0 & N^n & 0 \\ 0 & 0 & N^1 & \cdots & 0 & 0 & N^n \end{bmatrix} \tag{5.3}$$

The strain, $\boldsymbol{\varepsilon}$, has 6 components (no_strain=6) and may be derived from the discretized dof, $\mathbf{a}$, using the gradient matrix, $\mathbf{B}$,

$$\boldsymbol{\varepsilon} = \begin{bmatrix} \varepsilon_{11} \\ \varepsilon_{22} \\ \varepsilon_{33} \\ 2\varepsilon_{23} \\ 2\varepsilon_{13} \\ 2\varepsilon_{12} \end{bmatrix} = \mathbf{B}\,\mathbf{a} \tag{5.4}$$

where a block in the gradient matrix, $\mathbf{B} = [\,\mathbf{B}^1 \quad \mathbf{B}^2 \quad \cdots \quad \mathbf{B}^n\,]$ is defined as

$$\mathbf{B}^i = \begin{bmatrix} \partial N^i/\partial x_1 & 0 & 0 \\ 0 & \partial N^i/\partial x_2 & 0 \\ 0 & 0 & \partial N^i/\partial x_3 \\ 0 & \partial N^i/\partial x_3 & \partial N^i/\partial x_2 \\ \partial N^i/\partial x_3 & 0 & \partial N^i/\partial x_1 \\ \partial N^i/\partial x_2 & \partial N^i/\partial x_1 & 0 \end{bmatrix} \quad ; \quad i = 1, 2, ..., n \tag{5.5}$$

The stress, $\boldsymbol{\sigma}$, is found from the strain using the constitutive relation,

$$\boldsymbol{\sigma} = \begin{bmatrix} \sigma_{11} \\ \sigma_{22} \\ \sigma_{33} \\ \sigma_{23} \\ \sigma_{13} \\ \sigma_{12} \end{bmatrix} = \mathbf{C}\,\boldsymbol{\varepsilon} \tag{5.6}$$

A linear elastic material is defined by 2 parameters (Material.no_par=2): Youngs modulus, E, and Poisson's ratio, ν. For isotropic materials the constitutive matrix is

$$\mathbf{C} = \frac{E}{(1+\nu)(1-2\nu)} \begin{bmatrix} 1-\nu & \nu & \nu & 0 & 0 & 0 \\ \nu & 1-\nu & \nu & 0 & 0 & 0 \\ \nu & \nu & 1-\nu & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{2}(1-2\nu) & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{2}(1-2\nu) & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{1}{2}(1-2\nu) \end{bmatrix} \tag{5.7}$$

The element stiffness matrix has the same form as for potential elements, i.e.

$$\mathbf{K}^e = \int_\Omega \mathbf{B}^T\,\mathbf{C}\,\mathbf{B}\,d\Omega \tag{5.8}$$

but for solid mechanics it becomes a $3n \times 3n$ symmetric matrix. Calculation of the volume loads, $\mathbf{f}_v^e$, is dependent on the load field, $b(\mathbf{x})$. The field may be approximated from the node intensities, $\mathbf{b}^T = [\,b_1^1 \quad b_2^1 \quad b_3^1 \quad \cdots \quad b_1^n \quad b_2^n \quad b_3^n\,]$, using the shape functions, $\mathbf{N}(\mathbf{x})$,

$$\mathbf{b}(\mathbf{x}) = \mathbf{N}(\mathbf{x})\,\mathbf{b} \tag{5.9}$$

whereby the integral over the element becomes

$$\mathbf{f}_v^e = \int_\Omega \mathbf{N}^T(\mathbf{x})\,\mathbf{N}(\mathbf{x})\,d\Omega\;\mathbf{b} \tag{5.10}$$

For a surface, i, the surface traction, $\mathbf{t}^i(\mathbf{x})$, may be interpolated from the node intensities, $(\mathbf{t}^i)^T = [\,t_1^1 \quad t_2^1 \quad t_3^1 \quad \cdots \quad t_1^n \quad t_2^n \quad t_3^n\,]$, by use of the shape functions, i.e.

$$\mathbf{t}^i(\mathbf{x}) = \mathbf{N}(\mathbf{x})\,\mathbf{t}^i \tag{5.11}$$

The integral of the surface load, $\mathbf{f}_s^e$, consists of a sum of s surface integrals, s being the number of element surfaces, i.e.

$$\mathbf{f}_e^s = \sum_{i=1}^{s} \left[\int_{\Gamma_i} \mathbf{N}^T(\mathbf{x})\, \mathbf{N}(\mathbf{x})\, d\Gamma \right] \; \mathbf{t}^i \tag{5.12}$$

The difference between the integrals for linear elasticity and potential problems is that the number of dof is 3 instead of 1, thereby changing the form of shape function matrix and the gradient matrix.

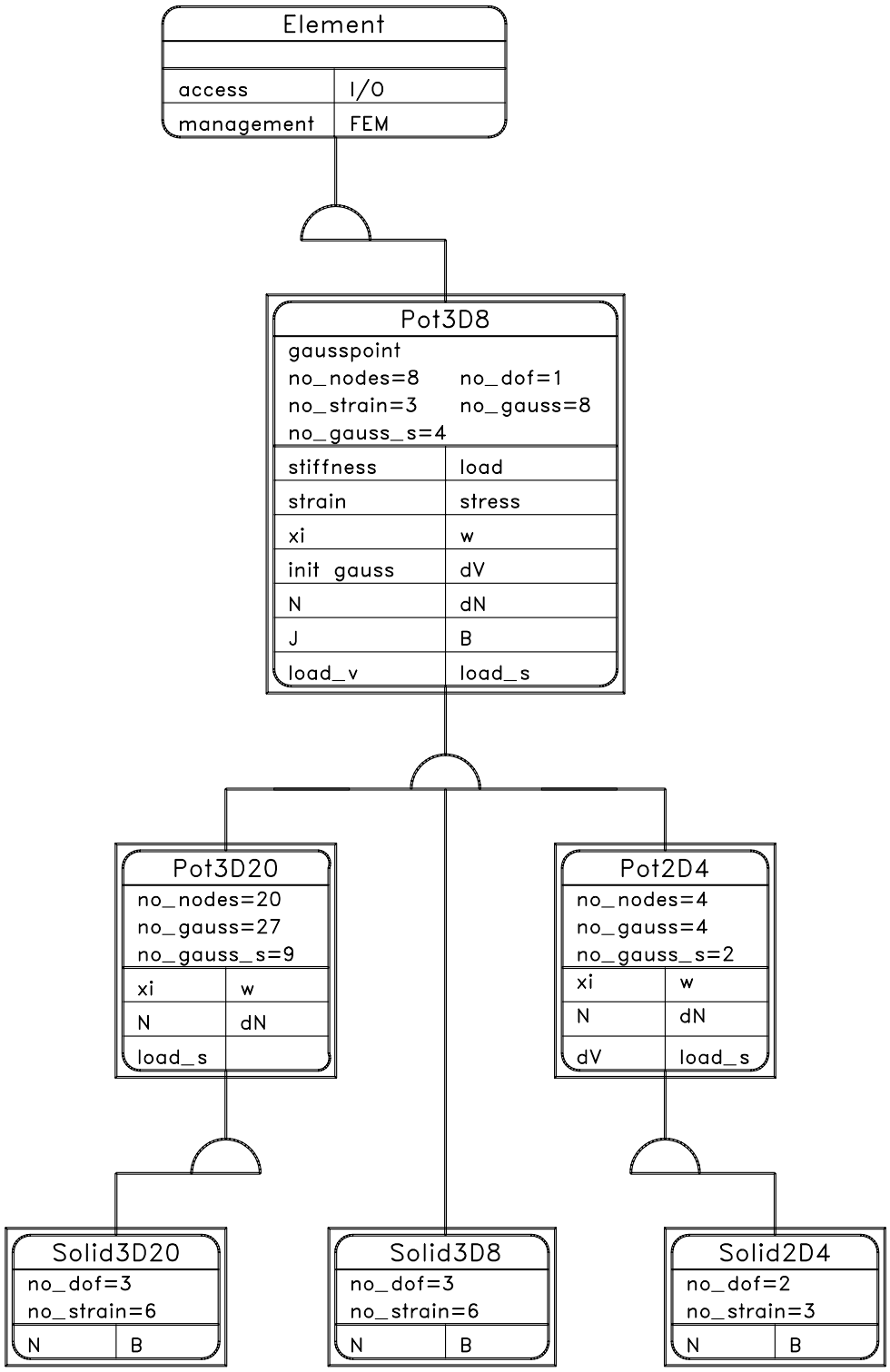


Figure 5.1: Continuum elements

5.2 Isoparametric solid element

The solid elements can also be defined as isoparametric elements. The similarities between potential problems and elasticity theory suggest that the solid elements should be derived from the potential elements. First of all, the elements uses the same shape functions and the same order of integration. The main differences are the expanded forms of the shape function matrix, $\mathbf{N}$, and the gradient matrix, $\mathbf{B}$. The components in these matrices is, however, exactly the same as for the potential problem, thus the redefined methods merely get the compact matrices from the potential methods and expand them to the correct format. The shape function matrix, $\mathbf{N}$ is set up in the following way,

N

```
Solid3D8.N(g)
   Matrix n(3,3*no_nodes)
   Nmat = Pot3D8.N(g)
   for (i=1 to no_nodes) do
     for (j=1 to no_dof) do
       n((i-1)*no_dof+j,j) = Nmat(i,j)
   return n
```

Similarly, the gradient matrix (5.5) is assembled from the compact matrix (4.20). Thus a solid element may inherit everything from the potential element redefining only the methods that set up the shape function matrix and the gradient matrix. The full hierarchy of continuum elements are presented in Figure 5.1.

5.3 Elastic materials

The solid elements uses a linear elastic, isotropic material model, as given by (5.7). This material model defines a class of elastic materials, with the 3D model as superclass. The 3D isotropic material is called Elastic. It is described by 2 parameters (no_par=2): Youngs modulus, E, is defined as par(1) while the Poisson ratio, ν, is par(2). The responsibility of each class is to set up the constitutive matrix, $\mathbf{C}$, and on request send it to the calling Element, which is handled by the implementation of C:

C

```
Elastic.C():
   Matrix c(6,6)
   E = par(1)
   nu = par(2)
   c(1,1) = c(2,2) = c(3,3) = 1–nu
   c(1,2) = c(2,1) = c(1,3) = c(3,1) = c(2,3) = c(3,2) = nu
   c(4,4) = c(5,5) = c(6,6) = 0.5 * (1–2*nu)
   denom = (1+nu) * (1–2*nu)
   c *= E/denom
   return c
```

Specializations to the plane problems, i.e. plane strain or plane stress, is done be over-

writing Elastic:C with the proper 2D definitions. For plane strain the constitutive matrix is

$$\mathbf{C} = \frac{E}{(1+\nu)(1-2\nu)} \begin{bmatrix} 1-\nu & \nu & 0 \\ \nu & 1-\nu & 0 \\ 0 & 0 & \frac{1}{2}(1-\nu) \end{bmatrix} \tag{5.13}$$

while for plane stress problems the constitutive matrix is

$$\mathbf{C} = \frac{E}{1-\nu^2} \begin{bmatrix} 1 & \nu & 0 \\ \nu & 1 & 0 \\ 0 & 0 & \frac{1}{2}(1-\nu) \end{bmatrix} \tag{5.14}$$

The hierarchy of elastic materials is presented in Figure 5.2.

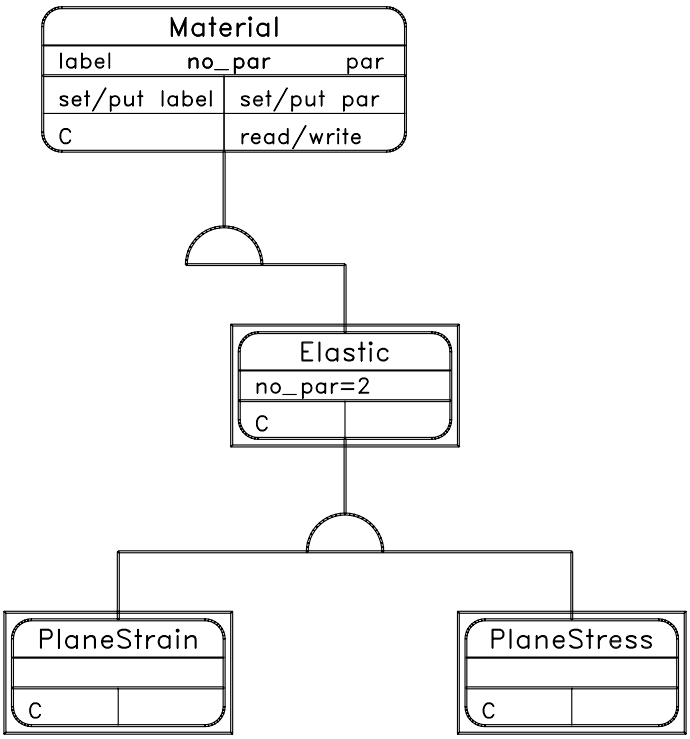


Figure 5.2: Elastic materials

Chapter 6

Non-linear finite elements

Finite elements establish the equilibrium equations as a relation between the external load, $\mathbf{f}$, and the internal force, $\mathbf{q}$, in terms of the generalized displacements, $\mathbf{a}$. Equilibrium can be stated as

$$\sum_{elements} \left[\int_{\Omega^e} \mathbf{B}^T \boldsymbol{\sigma} \, d\Omega \right] = \mathbf{f} \tag{6.1}$$

The left side represents the internal force, $\mathbf{q}$, that is usually expressed in terms of the discretized displacement, $\mathbf{a}$. The relation is generally a system of non-linear equations on the form

$$\mathbf{q}(\mathbf{a}) = \mathbf{f} \tag{6.2}$$

If the displacement is known the external load follow directly from (6.2), but usually it is the external load that is prescribed and the inverse relation must be determined. The relation between the displacement and the load is either linear or non-linear as illustrated in Figure 6.1. For linear problems the unknown displacement can be found explicitly by solving a system of linear equations, cf. Chapter 2,

$$\mathbf{K}\,\mathbf{a} = \mathbf{f} \tag{6.3}$$

In non-linear problems, where the relation between the displacement and the external load is not simple, it is necessary use an iterative strategy to solve the equations. This

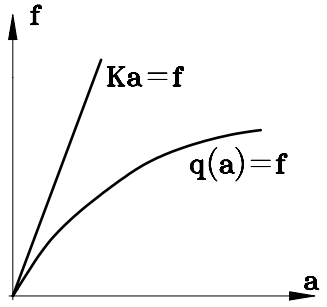


Figure 6.1: Linear and non-linear finite element problems

usually involves a series of load steps, $\Delta\mathbf{f}_1, \Delta\mathbf{f}_2, ...$, where the corresponding displacement increments, $\Delta\mathbf{a}_1, \Delta\mathbf{a}_2, ...$, are determined. Solution of non-linear finite element equations is a predictor-corrector strategy that for each load step consists of three parts: a prediction of the first displacement increment, a test of whether equilibrium is obtained and a strategy for correcting the first increment.

When an equilibrium point has been determined an additional load increment, $\Delta\mathbf{f}$, is applied. A first estimate on the displacement increment, $\Delta\mathbf{a}_1$, is found solving a linear tangent stiffness relation,

$$\mathbf{K}_t \, \Delta\mathbf{a}_1 = \Delta\mathbf{f} \tag{6.4}$$

The tangent stiffness matrix, $\mathbf{K}_t$, can be dependent on the current displacement, $\mathbf{a}$, the current stress, $\boldsymbol{\sigma}$ and the preceeding load history expressed by some state variables, $\boldsymbol{\alpha}$, i.e. $\mathbf{K}_t = \mathbf{K}_t(\mathbf{a}, \boldsymbol{\sigma}, \boldsymbol{\alpha})$. It is, however, often possible to distinguish between two types of non-linearity: geometrically non-linear problems and material non-linear problems. In geometrically non-linear problems it is the generalized gradient, $\boldsymbol{\nabla}$, that is non-linear in the displacement. The tangent stiffness thus becomes

$$\mathbf{K}_t(\mathbf{a}) = \sum_{elements} \left[\int_{\Omega^e} \mathbf{B}^T(\mathbf{a}) \mathbf{C} \, \mathbf{B}(\mathbf{a}) \, d\Omega \right] \tag{6.5}$$

In material non-linear problems it is the relation between the strain and stress that introduces the non-linearity. The material model is described by the constitutive matrix, $\mathbf{C}$, giving the following tangent stiffness

$$\mathbf{K}_t(\mathbf{a}, \boldsymbol{\sigma}, \boldsymbol{\alpha}) = \sum_{elements} \left[\int_{\Omega^e} \mathbf{B}^T \mathbf{C}(\mathbf{a}, \boldsymbol{\sigma}, \boldsymbol{\alpha}) \, \mathbf{B} \, d\Omega \right] \tag{6.6}$$

The evaluation of the tangent stiffness is part of the element formulation and will be described in the Chapters 7 and 8.

The next step in the solution procedure is to verify whether the estimate corresponds to an equilibrium state, (6.2). For this purpose the internal force has to be evaluated on basis of the current displacement estimate. The computation of the internal force can be divided in two types. In the first type of problems the internal force, $\mathbf{q}$, can be evaluated explicitly from the estimated displacement, $\mathbf{a}$. This type of problem includes non-linear strain measures, e.g.

$$\boldsymbol{\sigma} = \mathbf{C} \, \boldsymbol{\varepsilon}(\mathbf{a}) = \mathbf{C} \, \mathbf{B}(\mathbf{a}) \, \mathbf{a} \tag{6.7}$$

and non-linear, path-independent material models, such as non-linear elasticity, e.g.

$$\boldsymbol{\sigma} = \mathbf{C}(\boldsymbol{\varepsilon}) \, \boldsymbol{\varepsilon} \tag{6.8}$$

The strain, $\boldsymbol{\varepsilon}$, is found from the displacement estimate, i.e. $\boldsymbol{\varepsilon} = \mathbf{B}\mathbf{a}$, thus the internal force can be evaluated explicitly.

Path-dependent material models, such as elasto-plasticity, constitute the other class of non-linear problems. For such materials the stress in a point is dependent on the strain

and stress history and therefore it is not possible to express the constitutive behaviour in terms of total strain and stress. Instead an incremental formulation must be used. For rate-independent materials the relation between the strain increment, $d\boldsymbol{\varepsilon}$, and the stress increment, $d\boldsymbol{\sigma}$, is given by a tangent stiffness relation,

$$d\boldsymbol{\sigma} = \mathbf{C}(\boldsymbol{\sigma}, \boldsymbol{\varepsilon}, \boldsymbol{\alpha})\, d\boldsymbol{\varepsilon} \tag{6.9}$$

The incremental tangent stiffness, $\mathbf{C}$, depends on the current stress and strain, and on the stress history expressed in terms of some state variables, $\boldsymbol{\alpha}$. These are also path-dependent and the evolution of the state variables must be formulated on incremental form, as well. The internal force, $\mathbf{q}$, is found by integration of the total stress, $\boldsymbol{\sigma}$, over the elements. Because the constitutive behaviour is path-dependent the stress in a point is found by integrating the incremental relation, (6.9), and the state variables, $\boldsymbol{\alpha}$, over the complete load history. In practice this means that the load must be applied in a number of increments, each followed by equilibrium iterations. Within an iteration the finite stress increment, $\Delta\boldsymbol{\sigma}$, and the increment in the state variables, $\Delta\boldsymbol{\alpha}$, are evaluated from the estimated strain increment, $\Delta\boldsymbol{\varepsilon}$. Having obtained convergence the total stress and state variables are updated by their increments. Solution of non-linear problems with path-dependent material models thus consists of two iteration levels: the global equilibrium iterations that determines the displacement, $\mathbf{a}$, and iterations on point level within each element to integrate the stress, $\boldsymbol{\sigma}$, and state variables, $\boldsymbol{\alpha}$ for the estimated strain increment.

The evaluation of the internal force is part of the element formulation. Chapter 7 provides the formulation of a bar element with finite deformations as an example of a problem with explicit evaluation of the internal force. Chapter 8 deals with elasto-plastic material models where the internal force is obtained implicitly by integration over the complete load history.

If the estimate does not represent equilibrium there must be a strategy for correcting it. The residual, which is the unbalance between the internal force and the external load, is usually the main component in the strategy. The residual may be regarded as that part of the load that has not yet produced any displacement. Its contribution to the displacement increment is found by solving another linear stiffness relation. Additional provisions such as restrictions on the magnitude of displacement increment and modification of the external load can be employed to make the solution algorithm more robust. Evaluation of the residual and computation of the correction are termed equilibrium iterations. These are continued until the residual is smaller than a prescribed tolerance limit.

This chapter considers the solution of non-linear finite element problems. A general introduction to non-linear solution methods is given in order to identify two key concepts: equilibrium iterations and stiffness updates. The orthogonal residual algorithm, Krenk (1993b) and Krenk & Hededal (1993), is introduced. A pseudo-code describing an application for non-linear finite element problems summarizes the algorithm. Finally, additions to the Element class dictated by the non-linear solution strategies are presented.

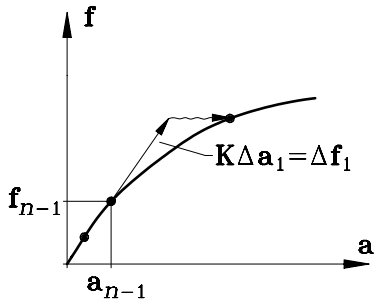


Figure 6.2: Solution of non-linear equations

6.1 Solution of non-linear finite element equations

The solution of non-linear finite element problems usually consists of a series of load steps, each involving iterations to establish equilibrium at the new load level. Each converged load step marks a point on the equilibrium path, Figure 6.2.

Assuming that an equilibrium point, $(\mathbf{a}_{n-1}, \mathbf{f}_{n-1})$, has been established, a new load increment, $\Delta\mathbf{f}_1$, is applied. A first estimate on the increment, $\Delta\mathbf{a}_1$, is obtained by solving a system of linear equations,

$$\mathbf{K}\,\Delta\mathbf{a}_1 = \Delta\mathbf{f}_1 \tag{6.10}$$

where $\mathbf{K}$ is a representative incremental stiffness matrix. Because the relation, (6.2), is non-linear the first increment does not represent an equilibrium state and iterations must be performed in order to determine the next point on the equilibrium path, see Figure 6.2. The iterations may involve changes in both the displacement and the load increments. For iteration i the increments, $\Delta\mathbf{a}_i$ and $\Delta\mathbf{f}_i$, are modified by the subincrements, $\delta\mathbf{a}_i$ and $\delta\mathbf{f}_i$, i.e.

$$\Delta\mathbf{a}_i = \Delta\mathbf{a}_{i-1} + \delta\mathbf{a}_i \tag{6.11}$$

$$\Delta\mathbf{f}_i = \Delta\mathbf{f}_{i-1} + \delta\mathbf{f}_i \tag{6.12}$$

If the estimate, $\mathbf{a}_{n-1} + \Delta\mathbf{a}_i$, does not satisfy the equilibrium equation, (6.2), there exists an unbalance between the internal force and the external load. This unbalance is termed the residual, $\mathbf{r}$, and is defined as

$$\mathbf{r}_i = \mathbf{f}_{n-1} + \Delta\mathbf{f}_i - \mathbf{q}(\mathbf{a}_{n-1} + \Delta\mathbf{a}_i) \tag{6.13}$$

The residual, $\mathbf{r}_i$, is usually the main component in the correction of the displacement increment in the following iteration. The residual, $\mathbf{r}_{i-1}$, thus gives the subincrement, $\delta\mathbf{a}_i$, that can be found by solving an incremental stiffness relation,

$$\mathbf{K}\,\delta\mathbf{a}_i = \mathbf{r}_{i-1} \tag{6.14}$$

where $\mathbf{K}$ is a representative stiffness matrix. The stiffness matrix can either be the same as used in (6.10) or it may be an updated matrix representing the current stiffness, as

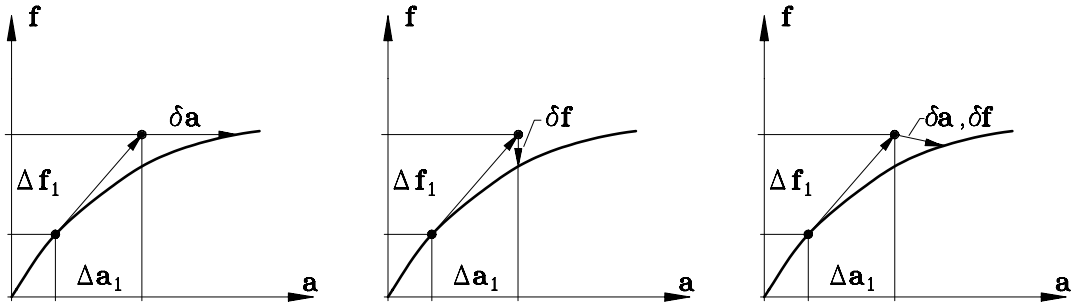


Figure 6.3: Equilibrium iterations: a) Newton-Raphson methods, b) Residual methods, c) Arc-length methods

discussed later on. The equilibrium iterations are carried on until the residual, $\mathbf{r}$, is lower than a given tolerance. Essential to this type of solution strategy is the residual vector and the stiffness matrix. The residual scales the subincrement, $\delta\mathbf{a}$, and the stiffness matrix determines the direction of the correction.

The equilibrium iterations may involve modifications of both the load and the displacement of as stated in (6.11) and (6.12). In the classical Newton-Raphson methods, Figure 6.3a, the load is kept constant during iterations and the displacement is then adjusted sequentially by evaluating the residual, (6.13), and the corresponding subincrement from (6.14). These methods have problems in passing load limit points, because the load is not adjusted. Another iteration strategy would be to adjust the load at fixed displacement, Figure 6.3b. For multi-dimensional problems the load can not be adjusted so that it exactly corresponds to an equilibrium point and therefore the displacement must also be modified. The correction of the displacement is found on basis of the residual from the modified load. In this type of iterations the corrections thus work in pair. The strategy, Figure 6.3b, is the basis for the residual methods, Bergan (1980,1981) and Krenk (1993b), where the external load is adjusted in order to optimize the residual, $\mathbf{r}$, used for calculating the subincrement, $\delta\mathbf{a}$, from (6.14). The orthogonal residual method is considered in detail in Section 6.2.

A third possibility is to adjust the load and displacement simultaneously, Figure 6.3c. Among the most popular of these methods are the arc-length methods, Riks (1979), Crisfield (1981), Ramm (1981). In the arc-length methods a constraint is imposed on the displacement and load increments, $\Delta\mathbf{a}$ and $\Delta\mathbf{f}$, such that they keep a constant 'arc-length' during all iterations. Within each iteration the subincrement, $\delta\mathbf{a}_i$, is first evaluated from (6.14) and then the updated increments, $\Delta\mathbf{a}_i$ and $\Delta\mathbf{f}_i$, are scaled to satisfy the constraint. Crisfield (1981) defines the constraint in terms of the Euclidian norm of the displacement increment, i.e.

$$||\Delta\mathbf{a}||_2 = l_{max} \tag{6.15}$$

where l_{max} is the prescribed arc-length. This constraint should be fulfilled for all increments during the equilibrium iterations. The first increment, $\Delta\mathbf{a}_1$, is found from the tangent relation, (6.10), with the load increment, $\Delta\mathbf{f}_1$, which is then adjusted so that the it fulfils

the constraint

$$\xi = \frac{l_{max}}{l} \tag{6.16}$$

The initial load factor ξ corresponds to a scaling of a linear estimate, thus the effective increments are $\xi\Delta\mathbf{a}_1$ and $\xi\Delta\mathbf{f}_1$. This estimate will in general not fulfil equilibrium, (6.2), and equilibrium iterations must performed. The displacement subincrement, $\delta\mathbf{a}$, is determined by a tangent relation, (6.14), where the residual is found from (6.13) with $\Delta\mathbf{f}_i = \xi\Delta\mathbf{f}_1$. The un-modified displacement increment, $\Delta\tilde{\mathbf{a}}$, then becomes

$$\Delta\tilde{\mathbf{a}} = \Delta\mathbf{a} + \delta\mathbf{a} \tag{6.17}$$

This increment may, however, violate the arc-length constraint, (6.15), and it must be corrected. A simple way to regain a correct arc-length is to modify the increment by a contribution along the tangent direction, $\Delta\mathbf{a}_1$, i.e.

$$\Delta\mathbf{a} = \Delta\tilde{\mathbf{a}} + \Delta\xi\Delta\mathbf{a}_1 \tag{6.18}$$

There are different ways to evaluate the correction factor, $\Delta\xi$. The constraint, (6.15), leads to a quadratic equation in $\Delta\xi$. Criteria for choice of the root is e.g. given by Crisfield (1981). As the correction is along the tangent direction it corresponds to a modification of the external load, $\xi\Delta\mathbf{f}_1$, thus the load factor is in each iteration updated by the correction factor, i.e.

$$\xi = \xi + \Delta\xi \tag{6.19}$$

The arc-length method is summarized in Algorithm 6.1. In order to make it a workable algorithm it must be supplemented criteria for reversing the load after passage of load limit points and for restarting in cases where convergence is not obtained within a maximum number of iterations. These issues are common for all algorithms and are considered in Section 6.2.2.

Another important part of a non-linear solution method is the strategy for updating the stiffness matrix, $\mathbf{K}$. In each equilibrium iteration the solution of the incremental stiffness relation, (6.14), requires a stiffness matrix and a residual vector. The residual vector must be formed in each iteration, but it is usually inexpensive to compute because it mainly involves vector operations on element level. Updating the stiffness matrix, however, may involve matrix manipulations on element level and may require the updated matrix to be factorized, both operations are expensive compared to the vector manipulation relating to the residual vector. Some strategies therefore updates the stiffness more rarely, e.g. in each equilibrium state, even though it may reduce the converge rate.

The Newton-Raphson methods are based on a first order Taylor expansion of the equilibrium equations. The stiffness matrix associated with this expansion is the tangent stiffness, $\mathbf{K}_t$, Figure 6.4a. In finite elements the tangent stiffness represents the stiffness associated with the current displacement state and is found by differentiation of the equilibrium equations, see e.g. Section 7.1. In a full Newton-Raphson scheme the stiffness

ALGORITHM 6.1: ARC-LENGTH METHOD - CRISFIELD (1981)

---

equilibrium state: $\mathbf{a}_{n-1}, \mathbf{f}_{n-1}$
load increment: $\Delta\mathbf{f}$
$\Delta\mathbf{a}_1 = \mathbf{K}_0^{-1}\Delta\mathbf{f}$
$\xi = l_{max}/\|\Delta\mathbf{a}_1\|_2$
$\Delta\mathbf{a} = \xi\Delta\mathbf{a}_1$
equilibrium iterations:
`do`
$\Delta\mathbf{q} = \mathbf{q}(\mathbf{a}_{n-1}+\Delta\mathbf{a})-\mathbf{f}_{n-1}$
$\mathbf{r} = -\Delta\mathbf{q} + \xi\Delta\mathbf{f}$
$\delta\tilde{\mathbf{a}} = \mathbf{K}_0^{-1}\mathbf{r}$
$\Delta\xi = \Delta\xi(\Delta\mathbf{a}, \delta\tilde{\mathbf{a}}, \Delta\mathbf{a}_1, l_{max})$
$\delta\mathbf{a} = \delta\tilde{\mathbf{a}} + \Delta\xi\Delta\mathbf{a}_1$
$\Delta\mathbf{a} = \Delta\mathbf{a} + \delta\mathbf{a}$
$\xi = \xi + \Delta\xi$
`until` $\|\mathbf{r}\| < \varepsilon\|\mathbf{f}\|$
update:
$\mathbf{a}_n = \mathbf{a}_{n-1} + \Delta\mathbf{a}$
$\mathbf{f}_n = \mathbf{f}_{n-1} + \xi\Delta\mathbf{f}$

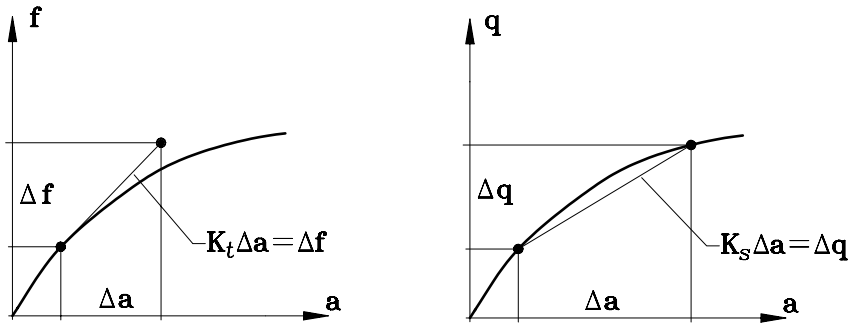


Figure 6.4: a) Tangent stiffness, b) Secant stiffness

matrix is evaluated for each state during the equilibrium iterations, while in the modified Newton-Raphson scheme the stiffness is evaluated only in equilibrium states.

The advantage of the full Newton-Raphson update is that it represents the current displacement state. It is, however, attractive only to calculate the tangent stiffness explicitly at equilibrium states mainly because it involves factorization. Alternatively, a quasi-Newton modification of the stiffness matrix could be used if it is observed the effective stiffness changes during equilibrium iterations. These modifications can be introduced directly in the factorized stiffness matrix and is therefore computationally inexpensive compared to a full Newton-Raphson update.

The idea in the quasi-Newton methods is to replace the original stiffness, $\mathbf{K}_0$, with a stiffness, $\mathbf{K}$, that exactly reproduces a known increment from a prescribed load increment. This quasi-Newton condition can be stated as a linear stiffness relation between the increment in the internal force, $\Delta\mathbf{q}$, and the displacement increment, $\Delta\mathbf{a}$, on the form

$$\mathbf{K}_s\,\Delta\mathbf{a} = \Delta\mathbf{q} \tag{6.20}$$

The stiffness matrix, $\mathbf{K}_s$, associated with the quasi-Newton condition represents a secant stiffness, Figure 6.4b. However, in multi-dimensional problems the secant is not uniquely defined and there exist various possibilities for choosing it. A standard technique for obtaining a correction in a direction, $\mathbf{v}$, consists of forming an exterior product with a vector, $\mathbf{w}$, i.e.

$$\mathbf{K}\,\mathbf{v} = (\mathbf{K}_0 + \mathbf{v}\mathbf{w}^T)\,\mathbf{v} = \mathbf{K}_0\,\mathbf{v} + \gamma\mathbf{v} \tag{6.21}$$

The correction along $\mathbf{v}$ is thus scaled by the projection of $\mathbf{v}$ onto $\mathbf{w}$. For finite element problems the BFGS update (Broyden-Fletcher-Goldfarb-Shanno) has been used with success. It consists of a symmetric rank-two correction of the stiffness matrix, i.e.

$$\mathbf{K}_s = \mathbf{K}_0 \;-\; \frac{(\mathbf{K}_0\Delta\mathbf{a})\,(\mathbf{K}_0\Delta\mathbf{a})^T}{\Delta\mathbf{a}^T\mathbf{K}_0\Delta\mathbf{a}} + \frac{\Delta\mathbf{q}\,\Delta\mathbf{q}^T}{\Delta\mathbf{q}^T\Delta\mathbf{a}} \tag{6.22}$$

where $\mathbf{K}_0$ is the stiffness corresponding to the last full update, e.g. the last equilibrium point. It is seen that if the stiffness, $\mathbf{K}_s$, is multiplied with the increment, $\Delta\mathbf{a}$, the first two terms on the right side in (6.22) cancel, i.e. the original stiffness is removed, leaving

only the increment in the internal force, $\Delta\mathbf{q}$. The BFGS update is attractive because the modifications can be introduced directly in the factored form of the global stiffness matrix, Matthies & Strang (1979). Thus, the quasi-Newton update does neither involve element calculations nor require the stiffness matrix to be factorized.

The two main ingredients in non-linear solution strategies are thus: equilibrium iterations and stiffness updates. The different techniques may be combined to form robust algorithms. Among the most popular are the arc-length methods of which there exists a number of variants. An interesting alternative to these methods are the orthogonal residual algorithms presented in the following.

6.2 The orthogonal residual method

An important part of a non-linear solution strategy is the ability to pass load limit points. This requires that both the displacement and the load can be adjusted during the equilibrium iterations. In the residual methods equilibrium iterations consist of an adjustment of the external load followed by a correction of the displacement increment. In each iteration the external load is adjusted at fixed displacement such that the residual becomes optimal. This optimal residual in then used to evaluate the correction.

Assume that an equilibrium point, $(\mathbf{a}_{n-1}, \mathbf{f}_{n-1})$, is established. An additional load increment, $\Delta\mathbf{f}$, is applied and equilibrium iterations are performed. In the current iteration the displacement estimate, $\mathbf{a}_{n-1} + \Delta\mathbf{a}$, corresponds to an internal force, $\mathbf{q}(\mathbf{a}_{n-1} + \Delta\mathbf{a})$. This force is generally not in equilibrium with the external load, $\mathbf{f}_{n-1} + \Delta\mathbf{f}$, but generates residual, $\mathbf{r}$. The load is adjusted by introducing the scaled load increment, $\xi\Delta\mathbf{f}$, instead of the original, $\Delta\mathbf{f}$. The residual from the scaled external load is

$$\mathbf{r} = \mathbf{f}_{n-1} + \xi\Delta\mathbf{f} - \mathbf{q}(\mathbf{a}_{n-1} + \Delta\mathbf{a}) = -\Delta\mathbf{q} + \xi\Delta\mathbf{f} \tag{6.23}$$

where $\Delta\mathbf{q}$ is the increment in the internal force from the last equilibrium state, $\mathbf{f}_{n-1} = \mathbf{q}_{n-1}$,

$$\Delta\mathbf{q} = \mathbf{q}(\mathbf{a}_{n-1} + \Delta\mathbf{a}) - \mathbf{q}_{n-1} \tag{6.24}$$

The scaling factor, ξ, is found from an optimality criterion. Bergan (1980,1981) stated optimality as a minimum condition,

$$\min_{\xi} \|\mathbf{r}\| = \min_{\xi} \|-\Delta\mathbf{q} + \xi\Delta\mathbf{f}\| \tag{6.25}$$

This minimum condition requires definition of an appropriate norm for the residual, e.g. in quadratic form

$$\|\mathbf{r}\|^2 = \mathbf{r}^T\,\mathbf{B}\,\mathbf{r} \tag{6.26}$$

where $\mathbf{B}$ is a symmetric positive definite matrix. Candidates for the $\mathbf{B}$ matrix could be the identity matrix, $\mathbf{I}$, leading to a Euclidian norm or the inverse of the stiffness matrix, $\mathbf{K}^{-1}$, which gives an energy norm. The choice of $\mathbf{B}$ is limited by the restriction that it must be positive definite. The energy norm is therefore difficult to use around limit points where

Figure 6.5: Minimum residual: a) Euclidian norm, b) **B**-norm

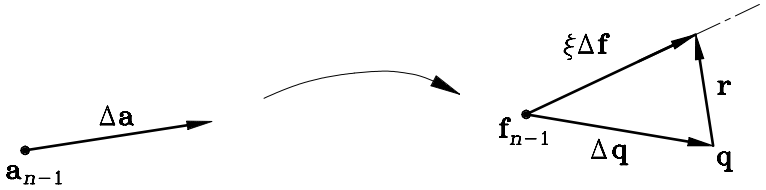


Figure 6.6: Orthogonal residual

the stiffness matrix, $\mathbf{K}$, is not positive definite. In either case the scaling factor may be determined explicitly by setting the derivative with respect to ξ equal to 0, i.e.

$$\Delta\mathbf{f}^T\,\mathbf{B}\,(-\Delta\mathbf{q}+\xi\Delta\mathbf{f})=0 \tag{6.27}$$

This relation states that the minimum residual, $\mathbf{r}$, should be $\mathbf{B}$-orthogonal to the increment in the external load, $\Delta\mathbf{f}$, see Figure 6.5. The scaling factor thus becomes

$$\xi=\frac{\Delta\mathbf{q}^T\,\mathbf{B}\,\Delta\mathbf{f}}{\Delta\mathbf{f}^T\,\mathbf{B}\,\Delta\mathbf{f}} \tag{6.28}$$

The subincrement, $\delta\mathbf{a}$, generated by the minimum residual is found from (6.14). Multiplying this relation with $(\mathbf{B}\Delta\mathbf{f})^T$ gives

$$(\mathbf{B}\Delta\mathbf{f})^T\mathbf{K}\,\delta\mathbf{a}=\Delta\mathbf{f}^T\mathbf{B}\,\mathbf{r}=0 \tag{6.29}$$

Thus all subincrements lie in a hyperplane with a normal vector, $\mathbf{K}^T\mathbf{B}\Delta\mathbf{f}$, provided the stiffness matrix is kept constant in all iterations. If this hyperplane does not intersect the true equilibrium path, e.g. at displacement limit point, the iterations will not converge.

An alternate optimality criterion stated in terms of conjugate variables is used by Krenk (1993b) to formulate an orthogonal residual method. In the orthogonal residual method the external load is assumed to be optimal when the residual, $\mathbf{r}$, will neither increase nor decrease the magnitude current displacement increment, $\Delta\mathbf{a}$. This optimality condition may be expressed as

$$\Delta\mathbf{a}^T\,\mathbf{r}=0 \tag{6.30}$$

It states that the external load is optimal when the residual is orthogonal to the current displacement increment, see Figure 6.6. It is noticed that this optimality condition is stated in conjugate variables and does not require any norm. Inserting the residual, (6.23), into the orthogonality condition, (6.30), gives

$$\Delta\mathbf{a}^T\,(-\Delta\mathbf{q}+\xi\Delta\mathbf{f})=0 \tag{6.31}$$

Comparing this relation with (6.27) it is seen that the term $\mathbf{B}\Delta\mathbf{f}$ is replaced by the current displacement increment, $\Delta\mathbf{a}$. If the energy norm is used, $\mathbf{B} = \mathbf{K}^{-1}$, the first iteration would be exactly the same for the two criteria, i.e. $\Delta\mathbf{a}_1 = \mathbf{K}^{-1}\Delta\mathbf{f}_1$. However, this identity is lost in the following iterations. Rearranging (6.31) gives the scaling factor,

$$\xi = \frac{\Delta\mathbf{q}^T\,\Delta\mathbf{a}}{\Delta\mathbf{f}^T\,\Delta\mathbf{a}} \tag{6.32}$$

Thus the scaling factor, ξ, is explicitly determined by evaluating two scalar products. The orthogonal residual, (6.23), is then used to evaluate the next subincrement, $\delta\mathbf{a}$. The direction of the subincrements can be investigated by multiplication of (6.14) with the current displacement increment, $\Delta\mathbf{a}$,

$$\Delta\mathbf{a}^T\mathbf{K}\delta\mathbf{a} = \Delta\mathbf{a}^T\mathbf{r} = 0 \tag{6.33}$$

It is seen that the subincrement, $\delta\mathbf{a}$, is $\mathbf{K}$-orthogonal to the current increment, $\Delta\mathbf{a}$. This means that the subincrements do not lock onto a fixed hyperplane as it is the case in the minimum residual method.

6.2.1 Dual orthogonality

The other part of a non-linear solution method relates to the stiffness update. In the orthogonal residual method the load adjustment is independent on the choice of stiffness update. In this framework the modified Newton-Raphson method has usually been employed, updating the stiffness matrix in each equilibrium point. Changes in the effective stiffness encountered during the equilibrium iterations are taken into account by a quasi-Newton modification which in combination with the orthogonality condition, (6.30), leads to a simple one-term correction of the displacement subincrement, Krenk & Hededal (1993).

The quasi-Newton modification applied in the following is the symmetric rank-two BFGS update, (6.22). The inverse of this secant stiffness matrix may be found by use of the Sherman-Morrison formula, see e.g. Luenberger (1984),

$$\mathbf{K}_s^{-1} = \left(\mathbf{I} - \frac{\Delta\mathbf{a}\,\Delta\mathbf{q}^T}{\Delta\mathbf{q}^T\Delta\mathbf{a}}\right)\mathbf{K}_0^{-1}\left(\mathbf{I} - \frac{\Delta\mathbf{q}\,\Delta\mathbf{a}^T}{\Delta\mathbf{q}^T\Delta\mathbf{a}}\right) + \frac{\Delta\mathbf{a}\,\Delta\mathbf{a}^T}{\Delta\mathbf{q}^T\Delta\mathbf{a}} \tag{6.34}$$

Using the modified stiffness matrix in the incremental stiffness relation, (6.14), yields

$$\delta\mathbf{a} = \mathbf{K}_s^{-1}\,\mathbf{r} \tag{6.35}$$

When the residual satisfies the orthogonality condition, (6.30), the terms involving products between the residual, $\mathbf{r}$, and the displacement increment, $\Delta\mathbf{a}$, vanishes leading to the following simple form of the incremental stiffness relation,

$$\delta\mathbf{a} = \left(\mathbf{I} - \frac{\Delta\mathbf{a}\,\Delta\mathbf{q}^T}{\Delta\mathbf{q}^T\Delta\mathbf{a}}\right)\mathbf{K}_0^{-1}\,\mathbf{r} \tag{6.36}$$

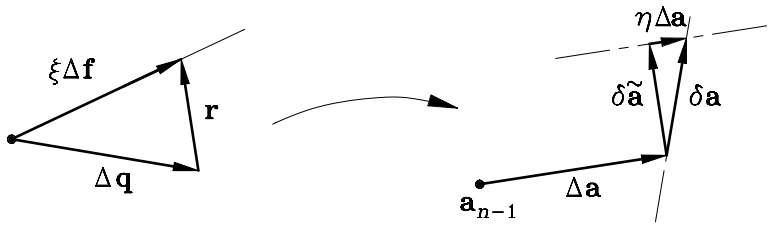


Figure 6.7: Quasi-Newton correction of a displacement subincrement

By introducing the subincrement without any quasi-Newton correction

$$\delta\tilde{\mathbf{a}} = \mathbf{K}_0^{-1}\,\mathbf{r} \tag{6.37}$$

the corrected subincrement becomes

$$\delta\mathbf{a} = \delta\tilde{\mathbf{a}} - \Delta\mathbf{a}\,\frac{\Delta\mathbf{q}^T\delta\tilde{\mathbf{a}}}{\Delta\mathbf{q}^T\Delta\mathbf{a}} = \delta\tilde{\mathbf{a}} + \eta\Delta\mathbf{a} \tag{6.38}$$

As for the load scaling factor, ξ, the displacement scaling factor, η, is found from two scalar products,

$$\eta = -\frac{\Delta\mathbf{q}^T\delta\tilde{\mathbf{a}}}{\Delta\mathbf{q}^T\Delta\mathbf{a}} \tag{6.39}$$

It is assumed that the magnitude of the current displacement increment, $\Delta\mathbf{a}$, is optimal for the current load level. This magnitude should be retained after the quasi-Newton correction, (6.38), and the total increment is divided by the factor, $1+\eta$, corresponding to a relaxation of the correction, i.e.

$$\delta\mathbf{a} = \frac{\delta\tilde{\mathbf{a}}}{1+\eta} \tag{6.40}$$

It is found that the quasi-Newton correction used in connection with the orthogonal residual leads to another orthogonality,

$$\Delta\mathbf{q}^T\delta\mathbf{a} = \Delta\mathbf{a}^T\mathbf{K}_s\,\delta\mathbf{a} = \Delta\mathbf{a}^T\mathbf{r} = 0 \tag{6.41}$$

This implies that any quasi-Newton modification that satisfies (6.20) will lead to a subincrement, $\delta\mathbf{a}$, that is orthogonal to the current increment in the internal force, $\Delta\mathbf{q}$. Thus the algorithm is characterized by a dual orthogonality: orthogonality between the residual and the total displacement increment and orthogonality between the increment in the internal force and the iterative subincrement.

The quasi-Newton modification consists in the orthogonal residual method of a one term correction of the the subincrement, $\delta\tilde{\mathbf{a}}$, along the current total increment, $\Delta\mathbf{a}$, see Figure 6.7. The modification does not involve matrix operations, thus is computationally efficient. Also it is notable that the quasi-Newton modification acts on the global equation system and does not require integrations on element level, like the tangent stiffness. The

ALGORITHM 6.2: ORTHOGONAL RESIDUAL METHOD

equilibrium state: $\mathbf{a}_{n-1}, \mathbf{f}_{n-1}$
load increment: $\Delta\mathbf{f}$
$\Delta\mathbf{a} = \mathbf{K}_0^{-1}\Delta\mathbf{f}$
equilibrium iterations:
`do`
$\Delta\mathbf{q} = \mathbf{q}(\mathbf{a}_{n-1}+\Delta\mathbf{a})-\mathbf{f}_{n-1}$
$\xi = \Delta\mathbf{q}^T\Delta\mathbf{a}/\Delta\mathbf{f}^T\Delta\mathbf{a}$
$\mathbf{r} = -\Delta\mathbf{q} + \xi\Delta\mathbf{f}$
$\delta\tilde{\mathbf{a}} = \mathbf{K}_0^{-1}\mathbf{r}$
$\eta = -\Delta\mathbf{q}^T\delta\tilde{\mathbf{a}}/\Delta\mathbf{q}^T\Delta\mathbf{a}$
$\delta\mathbf{a} = \delta\tilde{\mathbf{a}}/(1+\eta)$
$\Delta\mathbf{a} = \Delta\mathbf{a} + \delta\mathbf{a}$
`until` $\|\mathbf{r}\| < \varepsilon\|\mathbf{f}\|$
update:
$\mathbf{a}_n = \mathbf{a}_{n-1} + \Delta\mathbf{a}$
$\mathbf{f}_n = \mathbf{f}_{n-1} + \xi\Delta\mathbf{f}$

orthogonal residual method supplemented by the quasi-Newton correction is summarized in Algorithm 6.2. It is notable that this version of the orthogonal residual method only differs from the arc-length method, Algorithm 6.1, on two points: the direction of the correction of the subincrement and the determination of the load factor. In the orthogonal residual method the correction is in the direction of the current total increment, $\Delta\mathbf{a}$, while the arc-length method uses a correction along the tangent, $\Delta\mathbf{a}_1$. The load factor is in the orthogonal residual method evaluated independently of the displacement, namely from the orthogonality condition, whereas the arc-length follows directly from the correction of the displacement increment.

As for the arc-length method, Algorithm 6.1, the orthogonal algorithm must be supplemented with means for passage of load limit points and a restart strategy in case of slow convergence. This is considered in the following section.

6.2.2 Implementation of the orthogonal residual method

In order to develop a robust code on basis of the orthogonal residual method as presented in Algorithm 6.2 there are two issues that require attention: passage of load limit points and strategies for controlling the magnitude of the increments in regions with changes in the stiffness.

Having passed a load limit point the direction of the load increment is reversed, corresponding to a change of the sign on the load increment in the following load step. Continuing with the previous sign on the load increment would mean that the first displacement increment, $\Delta\mathbf{a}_1$, would have the wrong sign and double back along the equilibrium path, Figure 6.8a. Instead it must be ensured that the next step leads to a continuation of the equilibrium path, Figure 6.8b. If the stiffness does not change excessively, e.g. because of bifurcation, the first displacement increment, $\Delta\mathbf{a}_1$, is dominant and the iterative subincrements, $\delta\mathbf{a}$, mainly changes the magnitude of the final increment. The first displacement increment is therefore used to check if a load limit point has been passed. The projection of the first increment, $\Delta\mathbf{a}_1$, onto the final increment in the previous load step, $\Delta\mathbf{a}_{old}$, gives a simple condition,

$$\Delta\mathbf{a}_{old}^T\Delta\mathbf{a}_1 > 0 \tag{6.42}$$

If this condition is violated the sign of the displacement increment and the following load increments is changed.

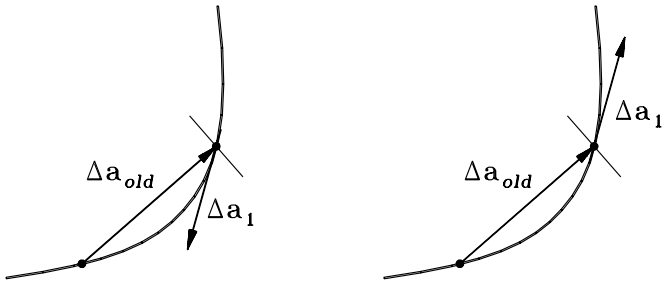


Figure 6.8: Direction control: a) doubling back, b) continuation of the equilibrium path

In regions with low stiffness, such as load limit points, the first displacement increment, $\Delta\mathbf{a}_1$, may become excessively large, Figure 6.9. However, a simple step length control handles the problem. Instead of evaluating the residual from the out-of-scale value the increment is scaled back along its direction,

$$\Delta\tilde{\mathbf{a}}_1 = \rho\,\Delta\mathbf{a}_1 \tag{6.43}$$

This adjustment of the first increment is done before starting the equilibrium iterations and does therefore not violate the orthogonality condition, (6.30). The scaling factor, ρ, is defined as

$$\rho = \min\left(1, \frac{l_{max}}{\|\Delta\mathbf{a}_1\|}\right) \tag{6.44}$$

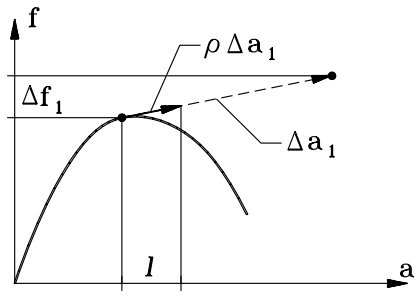


Figure 6.9: Scaling of first increment

The norm could be the maximum norm, $\|\cdot\|_\infty$, which scales the increment relative to the maximum displacement component. Another possibility is to use the Euclidian norm, $\|\cdot\|_2$, which is a scaling in the multi-dimensional displacement space. The relation, (6.44), uses a norm expressed only in terms of the displacement. Tracing an equilibrium path gives of a number of displacement states, $\mathbf{a}_1, \mathbf{a}_2, ...$ To obtain a good representation of the entire path the distance between these points should be kept relatively constant. As the load may change arbitrarily from point to point, a norm that includes the load could lead to arbitrary scaling factors. It should be noticed that the scaling factor, ρ, is identical with the initial load factor, ξ, of the arc-length method, Algorithm 6.1. The Euclidian scaling thus shows similarity with the version of the arc-length method presented by Crisfield (1981).

The step length restriction, (6.43), should only be active in regions with low stiffness. In these regions the algorithm changes from load control to displacement control. The choice of the maximum step length, l_{max}, determines when this change takes place. In the present framework an absolute maximum step length, l_{abs}, is defined relative to the initial stiffness. The first increment in the following load steps can not exceed some constant, C, times the size of the first increment in the first load step, $\Delta\mathbf{a}_{init}$, i.e.

$$l_{abs} = C\,\|\Delta\mathbf{a}_{init}\| \tag{6.45}$$

where the norm should be the same as used in (6.43). The algorithm thus changes to displacement control when the stiffness becomes less than C^{-1} of the initial stiffness.

In regions where the stiffness changes rapidly the first displacement increment, $\Delta\mathbf{a}_1$, which is based on the stiffness at the beginning of the iterations, may have a direction that is not representative for the final convergent increment. Thus, if the first increment is too long the iterations may not converge within a maximum number of iterations, see Figure 6.10a. It may therefore be necessary locally to restart the iterations from the previous equilibrium point with a new first increment. In this framework the restart procedure, which is evoked if convergence is not obtained within i_{max} iterations, sets the new first increment to half the previous one and performs new equilibrium iterations on basis of this estimate, see Figure 6.10b. This procedure is continued until convergence is reached, thus after m reductions the resulting first increment is

$$\Delta\mathbf{a}_1 = \psi\Delta\mathbf{a}_{start} \quad , \quad \psi = 0.5^m \tag{6.46}$$

where $\Delta\mathbf{a}_{start}$ is the first increment evaluated at the beginning of the load step.

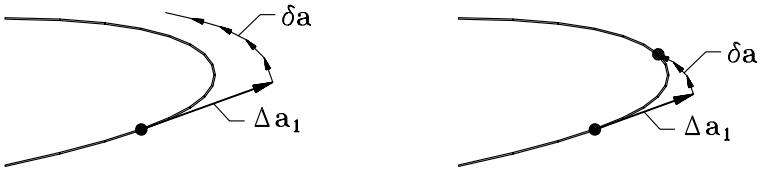


Figure 6.10: Restart: a) Full first increment, b) Reduced first increment

Having reached convergence a new load increment is applied. It is, however, convenient to keep the reduced increment until it is observed that the direction of the first increment again is representative for the final increment. This reduction is imposed via the step length restriction, (6.43). The maximum step length, l_{max}, in the load step following a reduction is reduced by the same factor, ψ, as the first increment, i.e.

$$l_{max} = \psi l_{max} \tag{6.47}$$

This maximum step length is kept until the first increment again becomes dominant. The number iterations, i, that is used to obtain convergence may be used as an indicator. If the number of iterations, i, in the previous load step is lower than a desired number of iterations, i_d, the maximum step length is doubled, i.e.

$$l_{max} = \min(l_{abs}, 2l_{max}) \tag{6.48}$$

Thus it is assured that the first increment can not exceed the absolute maximum, l_{abs}. In connection with the arc-length method a continuous modification of the step length is often used, see e.g. Crisfield (1981). The new maximum step length is scaled relative to the ratio of the desired number of iterations to the number of iterations used in the previous load step, i.e.

$$l_{max} = \min\left(l_{abs}, \left(\frac{i_d}{i}\right)^{\alpha} l_{max}\right) \tag{6.49}$$

where the exponent, α, is usually chosen in the interval $[0.5; 2.0]$.

The full implementation of the orthogonal residual algorithm is presented as Algorithm 6.3. It uses modified Newton-Raphson stiffness update that can be supplemented by the quasi-Newton correction, (6.38). The quasi-Newton correction is evoked if the boolean QN is TRUE. Convergence is measured using a reduced Euclidian norm where only the free force components are included, i.e.

reduced norm

```
rnorm(f):
   for (i=1 to no_dof) do
      if (fix(i) = FALSE)
         norm += sqr(f(i))
   return sqrt(norm)
```

Convergence is obtained if

$$\|\mathbf{r}\|_{red} \;<\; \varepsilon \, \|\Delta \mathbf{f}\|_{red} \tag{6.50}$$

where ε is the convergence threshold.

If convergence is not obtained within i_max iterations the restart procedure is evoked. It sets a boolean new_step equal to FALSE, whereby a stiffness update and determination of a new first increment are omitted. Instead the increment, da, is found as the reduction factor, psi, times the first increment, da_start, which is saved in the beginning of a new load step. When convergence is obtained the boolean new_step is set TRUE, thus telling the algorithm to update the stiffness and determine a new first increment. The maximum step length, l_max, is reduced by the factor, psi, before psi is reset to 1. The total dof, a, and load, fe, is then updated by their converged increments, da and xi*df.

The provisions made to make the orthogonal residual algorithm workable can be used to formulate an application of the arc-length method as well. The changes, however, may simply be introduced as options in the orthogonal residual algorithm, Algorithm 6.3.

It should be noted that the programming of the global solution algorithm is entirely done on application level, cf. Section 3.5, once the Element class has provided a representative tangent stiffness and the internal force.

ALGORITHM 6.3: APPLICATION FOR SOLUTION OF NON-LINEAR EQUATIONS

```
initialize:
a = da = fe = 0
psi = 1
new_step = TRUE
for (n=1 to no_loadstep) do
begin load step
  update stiffness and find first increment:
  if (new_step = TRUE)
  {
    K = 0
    for (elem=ellist.start() to ellist.end) do
      elem.assm_stiffness(K)
    factor(K,fix)
    da_old = da
    solve(K,da,df,fix)
    if (dot(da_old,da) < 0)
      df = -df
      da = -da
    l = norm(da)
    if (n=1)
      l_abs = l_max = c * l
    if (i < i_d)
      l_max = min(l_abs,2*l_max)
    rho = min(1,l_max/l)
    da_start = da
    da *= rho
    if (ARC = TRUE) xi = rho
  }
```

ALGORITHM 6.3: (*continued*)

```
    equilibrium iterations:
  i = 0
  do
  {
    i += 1
    fi = 0
    for (elem=ellist.start() to ellist.end) do
      elem.assm_intforce(fi,a+da)
    dq = fi – fe
    if (OR = TRUE) xi = dot(dq,da)/dot(df,da)
    r = –dq + (xi * df)
    solve(K,delta,r,fix)
    if (OR = TRUE and QN = TRUE)
      eta = –dot(dq,delta)/dot(dq,da)
      delta /= (1 + eta)
    else if (ARC = TRUE)
      dxi = dxi(da,da_start,delta,l_max)
      delta += dxi * da_start
      xi += dxi
    da += delta
  }
  until (rnorm(r)<EPS*rnorm(df) or i>i_max)
  restart:
  if (i>i_max)
  {
    new_step = FALSE
    psi *= 0.5
    da = psi * rho * da_start
    if (ARC = TRUE) xi = rho * psi
  }
  update:
  else
  {
    new_step = TRUE
    l_max *= psi
    psi = 1
    a += da
    fe += xi * df
    for (node=nolist.start() to nolist.end) do
      node.get_disp(a)
      node.get_load(fe)
  }
end load step
```

6.3 Extensions to the Element class

A solution strategy based on (6.14) requires that the Element must be able to evaluate the internal force and the current stiffness from the total displacement, **a**. There are different ways to obtain this: either Element methods take the total displacement as argument or they retrieve the updated values from the Nodes.

ObjectFEM usually uses a modified Newton-Raphson stiffness update evaluating the stiffness matrix at each equilibrium points. For each equilibrium state the Nodes store the converged displacement. The Element can thus retrieve the displacement by the get_dof method and evaluate the stiffness. In this way the initial declaration of stiffness, where it does not take arguments, is retained.

The internal force is always evaluated on basis of the current displacement estimate, which may not correspond to equilibrium. Retrieving the displacement from the Nodes would mean that these should store non-converged increments and provide a method that returns the updated displacement to the Element. In ObjectFEM the Element methods, assm_intforce and intforce, take the global displacement vector as argument, thus avoiding addition of attributes and methods to the Node. Solution of non-linear problems adds only two methods to the existing linear Element class: assm_intforce and intforce. The first is a static method that takes care of the extraction of the element displacement from the global displacement vector and after evaluation of the internal force assm_intforce assembles the updated values in the global vector containing the internal force, i.e.

assm intforce

```
Element.assm_intforce(gl_fint,a)
   size = no_dof * no_nodes
   for (i=1 to size) do
      dof(i) = a(dofno(i))
   fint = intforce(dof)
   for (i=1 to size) do
      gl_fint(dofno(i)) += fint(i)
```

The method intforce is a virtual method, that must be defined for elements that uses either a non-linear strain measure or a non-linear material model, but also for linear elements that are used in a non-linear analysis. The extended Element class is illustrated in Figure 6.11.

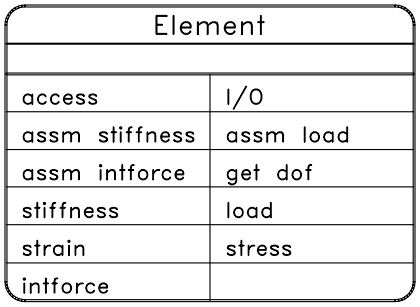


Figure 6.11: Extension of Element to non-linear problems

Chapter 7

Bar elements

The solid mechanics element presented in Chapter 5 may be used in all types of structural analysis, e.g. describing bars, beams, plates or shells. In classical structural mechanics the continuum formulation is condensed into specialized theories such as beam theory or plate theory depending on the shape and loading conditions of the structural member. Using these theories the number of degrees-of-freedom is reduced and it is possible in special cases to obtain exact solutions to the differential equations. Special types of elements are based on such theories. Unlike the potential elements and the continuum elements, they give exact solutions to the differential equations and are an efficient way to analyze more complicated structures using the classical theories. The elements are born on discrete form, i.e. the analytical solutions dictate the degrees-of-freedom and choice of shape functions, hence the stiffness matrix and the strain measure can be evaluated explicitly.

The bar element is one of these. This chapter presents an elastic bar element with finite deformation which uses a non-linear strain measure. This bar element represents a non-linear problem where the internal force can be evaluated explicitly from the estimated displacement, cf. Chapter 6.

First, the equilibrium equations for an elastic bar are formulated. This establishes the internal force in terms of the current displacement increment. Differentiation of the equilibrium equations leads a tangent stiffness that can be used in the global solution algorithm. From the non-linear formulation the linear part can be extracted to give a neat formulation of a geometrically linear bar element, implemented as class Bar. The non-linear bar element class, NlBar, is derived from Bar. In order to use NlBar element in a non-linear analysis the stiffness method is modified so that it represents the tangent stiffness and the internal force is evaluated by the method intforce. The chapter is concluded by examples where the orthogonal residual method and the arc-length method, Algorithm 6.3, are used for tracing the equilibrium path of non-linear truss structures.

7.1 Elastic bar element with finite deformations

In this section a bar element with finite deformations based on the Green strain measure is formulated. The purpose is to obtain expressions for the internal force and a tangent stiffness. The formulation is given on matrix form and applies to both two- and three-

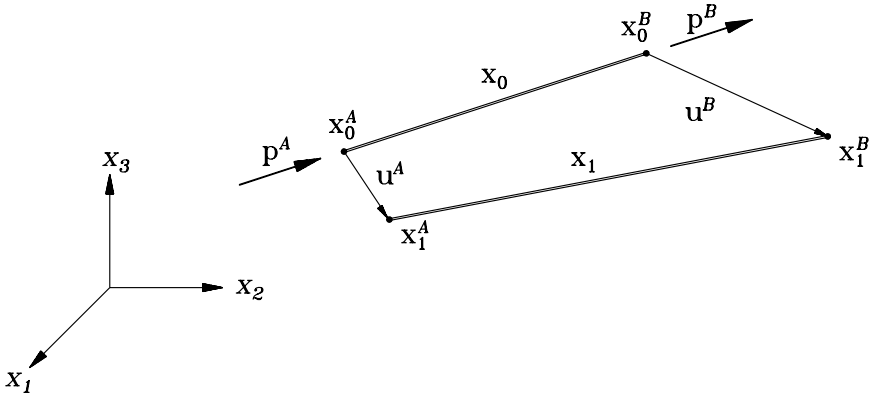


Figure 7.1: Elastic bar

dimensional problems. The presentation follows Krenk (1993a).

Consider an elastic bar with 2 nodes, A and B, as shown in Figure 7.1. The initial state is described by a directional vector, $\mathbf{x}_0$, defined from the nodal position vectors,

$$\mathbf{x}_0 = \mathbf{x}_0^B - \mathbf{x}_0^A \tag{7.1}$$

Subjecting the bar to 2 end forces, $\mathbf{p}^A$ and $\mathbf{p}^B$, produces displacements in each node, $\mathbf{u}^A$ and $\mathbf{u}^B$. The state of the bar is, however, only affected by the difference in the nodal displacements, thus a deformation vector, $\mathbf{u}$, is introduced as

$$\mathbf{u} = \mathbf{u}^B - \mathbf{u}^A \tag{7.2}$$

Thereby the deformed bar can be described by a directional vector, $\mathbf{x}_1$,

$$\mathbf{x}_1 = \mathbf{x}_1^B - \mathbf{x}_1^A = \mathbf{x}_0 + \mathbf{u} \tag{7.3}$$

For bars undergoing finite deformations it is necessary to use a non-linear strain measure. Here the Green strain is used,

$$\varepsilon_G = \frac{l_1^2 - l_0^2}{2l_0^2} \tag{7.4}$$

where l_0 is the initial length of the bar and l_1 is the length of the deformed bar. Inserting the vector representations of the undeformed and the deformed bar yields

$$\varepsilon_G = \frac{\mathbf{x}_1^T\mathbf{x}_1 - \mathbf{x}_0^T\mathbf{x}_0}{2l_0^2} = \frac{\left(\mathbf{x}_0 + \frac{1}{2}\mathbf{u}\right)^T\mathbf{u}}{l_0^2} \tag{7.5}$$

in which (7.3) is used to rewrite $\mathbf{x}_1$. The total Green strain, ε_G, is found as a projection of the displacement, $\mathbf{u}$, onto the intermediate direction vector, $\mathbf{x}_0 + \frac{1}{2}\mathbf{u}$. Taking the first variation of this gives

$$\delta\varepsilon_G = \frac{1}{l_0^2}\left(\mathbf{x}_0 + \mathbf{u}\right)^T\delta\mathbf{u} = \frac{1}{l_0^2}\mathbf{x}_1^T\delta\mathbf{u} \tag{7.6}$$

The increment in the Green strain, $\delta\varepsilon_G$, is found as the projection of the displacement increment, $\delta\mathbf{u}$, onto the current direction vector, $\mathbf{x}_1$. Assuming that there exists an axial force, N, which is conjugate to the increment in the Green strain, $\delta\varepsilon_G$, makes it possible to state a virtual work, δV. The internal force, N, works through the strain increment, while the external load, $\mathbf{p}$, works through the nodal displacement increments, i.e.

$$\begin{aligned}\delta V &= \int_{l_0} N\,\delta\varepsilon_G\,ds - (\mathbf{p}^A)^T\delta\mathbf{u}^A - (\mathbf{p}^B)^T\delta\mathbf{u}^B \\ &= \int_{l_0} \frac{N}{l_0^2}\mathbf{x}_1^T\delta\mathbf{u}\,ds - (\mathbf{p}^A)^T\delta\mathbf{u}^A - (\mathbf{p}^B)^T\delta\mathbf{u}^B = 0\end{aligned} \tag{7.7}$$

Dividing the internal work in contributions from $\delta\mathbf{u}^A$ and $\delta\mathbf{u}^B$ enables (7.7) to be written as

$$-\left(\int_{l_0}\frac{N}{l_0^2}\mathbf{x}_1\,ds + \mathbf{p}^A\right)^T\delta\mathbf{u}^A + \left(\int_{l_0}\frac{N}{l_0^2}\mathbf{x}_1\,ds - \mathbf{p}^B\right)^T\delta\mathbf{u}^B = 0 \tag{7.8}$$

As this must hold for arbitrary displacement integration yields

$$\mathbf{p}^A = -\mathbf{p}^B = -\frac{N}{l_0}\mathbf{x}_1 \tag{7.9}$$

This reveals that the true axial force in the deformed bar is not the conjugate force, N, but the scaled internal force, $(l_1/l_0)N$. This axial force balances 2 end forces that in the deformed state are aligned with the deformed bar, $\mathbf{x}_1$.

For linear elastic materials the conjugate force, N, is found from the strain,

$$N = EA\,\varepsilon_G \tag{7.10}$$

where E is Youngs modulus and A is the cross section area of the bar. It is notable that even though a linear constitutive model is used, the force, N, is non-linear with respect to the deformation, $\mathbf{u}$, through the Green strain. Using the constitutive relation (7.9) becomes

$$\mathbf{p}^A = -\mathbf{p}^B = -\varepsilon_G\frac{EA}{l_0}\mathbf{x}_1 \tag{7.11}$$

This is the balance equation establishing the relation between external load, $\mathbf{p}$, and the internal force. Thus the balance equation, (7.11), is found from the weak formulation, (7.7). It is interesting to notice the resemblance between (7.7) and (2.11), thus the formulation given in Chapter 2 is able to capture the effect of a non-linear strain measure. The internal force given as equivalent nodal forces can be evaluated from (7.11). This defines the evaluation of the internal force in terms of the strain, ε_G, which is given as a non-linear measure of the displacement - the dof.

7.1.1 Tangent stiffness

The solution strategies also requires the element to give a representative stiffness measure for the current state. The tangent stiffness is a commonly used measure and it can be found by differentiation of the balance equation, (7.9),

$$d\mathbf{p}^A = -d\mathbf{p}^B = -\left(\frac{\mathbf{x}_1}{l_0}dN + \frac{N}{l_0}d\mathbf{u}\right) = -\left(\frac{\mathbf{x}_1}{l_0}\frac{dN}{d\mathbf{u}} + \frac{N}{l_0}\mathbf{I}\right)d\mathbf{u} \tag{7.12}$$

The first term in (7.12) relates to the change in the internal force, dN, appearing if the bar is deformed. The last term appears due to a change in the direction, e.g. a rigid body rotation. It is related to the initial stress, N, which may be introduced as prescribed prestressing or due to a deformation of the bar. By introducing the constitutive equation, (7.10), and (7.5) the first term is rewritten as

$$\frac{dN}{d\mathbf{u}} = \frac{EA}{l_0}\frac{d\varepsilon_G}{d\mathbf{u}} = \frac{EA}{l_0^2}\mathbf{x}_1^T \tag{7.13}$$

whereby (7.12) can be given as

$$d\mathbf{p}^A = -d\mathbf{p}^B = -\left(\frac{\mathbf{x}_1\mathbf{x}_1^T}{l_0^3}EA + \frac{N}{l_0}\mathbf{I}\right)d(\mathbf{u}^B - \mathbf{u}^A) \tag{7.14}$$

On matrix form this is

$$\begin{bmatrix} d\mathbf{p}^A \\ d\mathbf{p}^B \end{bmatrix} = \left(\frac{EA}{l_0^3}\begin{bmatrix} \mathbf{x}_1\mathbf{x}_1^T & -\mathbf{x}_1\mathbf{x}_1^T \\ -\mathbf{x}_1\mathbf{x}_1^T & \mathbf{x}_1\mathbf{x}_1^T \end{bmatrix} + \frac{N}{l_0}\begin{bmatrix} \mathbf{I} & -\mathbf{I} \\ -\mathbf{I} & \mathbf{I} \end{bmatrix}\right)\begin{bmatrix} d\mathbf{u}^A \\ d\mathbf{u}^B \end{bmatrix} \tag{7.15}$$

This establishes the balance equations on incremental form with the tangent stiffness, $\mathbf{K}_t$, describing a linear relation between the increments in the external load and the increment in the node displacement:

$$d\mathbf{f} = \mathbf{K}_t\, d\mathbf{a} \tag{7.16}$$

where the 2 vectors are

$$d\mathbf{f} = \begin{bmatrix} d\mathbf{p}^A \\ d\mathbf{p}^B \end{bmatrix} \quad ; \quad d\mathbf{a} = \begin{bmatrix} d\mathbf{u}^A \\ d\mathbf{u}^B \end{bmatrix} \tag{7.17}$$

The tangent stiffness can be used as a linear predictor by the non-linear solution algorithms. It can be divided into 3 parts: linear stiffness, $\mathbf{K}_0$, initial displacement stiffness, $\mathbf{K}_u$ and initial stress stiffness, $\mathbf{K}_\sigma$, i.e.

$$\mathbf{K}_t = \mathbf{K}_0 + \mathbf{K}_u + \mathbf{K}_\sigma \tag{7.18}$$

where

$$\mathbf{K}_0 = \frac{EA}{l_0^3}\begin{bmatrix} \mathbf{x}_0\mathbf{x}_0^T & -\mathbf{x}_0\mathbf{x}_0^T \\ -\mathbf{x}_0\mathbf{x}_0^T & \mathbf{x}_0\mathbf{x}_0^T \end{bmatrix} \tag{7.19}$$

Table 7.1: Concepts for elastic bars

Initial state:	$\mathbf{x}_0$
Deformed state:	$\mathbf{x}_1$
Deformation:	$\mathbf{u}$
Strain:	ε_G
Stress:	N
Internal force:	$\frac{N}{l_0}\mathbf{x}_1$
External load:	$\mathbf{p}$
Constitutive model:	$N = \varepsilon_G EA$
Dof:	$\mathbf{a}$
Load:	$\mathbf{f}$
Tangent stiffness:	$\mathbf{K}_t = \mathbf{K}_0 + \mathbf{K}_u + \mathbf{K}_\sigma$

is the stiffness corresponding to a linear strain measure. The initial displacement stiffness contains the contribution that follows from using a non-linear strain measure and has the form,

$$\mathbf{K}_u = \frac{EA}{l_0^3}\begin{bmatrix} \mathbf{x}_0\mathbf{u}^T + \mathbf{u}\mathbf{x}_0^T + \mathbf{u}\mathbf{u}^T & -(\mathbf{x}_0\mathbf{u}^T + \mathbf{u}\mathbf{x}_0^T + \mathbf{u}\mathbf{u}^T) \\ -(\mathbf{x}_0\mathbf{u}^T + \mathbf{u}\mathbf{x}_0^T + \mathbf{u}\mathbf{u}^T) & \mathbf{x}_0\mathbf{u}^T + \mathbf{u}\mathbf{x}_0^T + \mathbf{u}\mathbf{u}^T \end{bmatrix} \tag{7.20}$$

The initial stress stiffness is identified as the last term in (7.15),

$$\mathbf{K}_\sigma = \frac{N}{l_0}\begin{bmatrix} \mathbf{I} & -\mathbf{I} \\ -\mathbf{I} & \mathbf{I} \end{bmatrix} \tag{7.21}$$

The initial stress stiffness arises, as it is seen from (7.12), because the internal force, N, must change its direction.

7.1.2 Total and updated Lagrangian formulation

The representation of the bar element presented above relates the state of the bar to an initial configuration, $\mathbf{x}_0$, thus a Lagrangian formulation is used. The solution of non-linear problems involves a number of load steps, $\Delta\mathbf{f}_1, \Delta\mathbf{f}_2, ...$, in which deformed configurations, $\mathbf{x}_1, \mathbf{x}_2, ...$, are determined. This gives 2 possibilities for the choice of reference configuration. In the total Lagrangian formulation the initial configuration, $\mathbf{x}_0$, is retained as reference. In this case the tangent stiffness must include all 3 terms in (7.18). In an updated Lagrangian formulation the previous equilibrium configuration, $\mathbf{x}_{n-1}$, is used as reference. In this case

the tangent stiffness does not contain the initial displacement stiffness as $\mathbf{u} = \mathbf{0}$. The tangent stiffness thus becomes

$$\mathbf{K}_t = \mathbf{K}_0 + \mathbf{K}_\sigma \tag{7.22}$$

where the linear stiffness is evaluated replacing $\mathbf{x}_0$ with the current configuration, $\mathbf{x}_{n-1}$. The initial stress stiffness requires the internal force, N, to be evaluated. This is done from the total strain in the current configuration,

$$N_{n-1} = EA\,\varepsilon_G(\mathbf{x}_0, \mathbf{u}_{n-1}) = EA\,\frac{\left(\mathbf{x}_0 + \frac{1}{2}\mathbf{u}_{n-1}\right)^T \mathbf{u}_{n-1}}{l_0^2} \tag{7.23}$$

It is noted that using the updated Lagrangian formulation simply consists of adding an initial stress contribution to the stiffness of a linear bar element described in the updated configuration. Still, the strain and consequently also the internal force is scaled relative to the initial length, l_0. If instead the current length, l_{n-1}, is used as scaling factor in an updated Lagrangian formulation, the solution becomes sensitive to the load increment size, see e.g. Yang & Leu (1991).

7.2 Linear bar element

From the derivation of the non-linear bar element emerges a neat formulation of the linear bar element. Neglecting the non-linear terms the linear stiffness, $\mathbf{K}_0$, represents the stiffness of a bar with a linear strain measure,

$$\varepsilon = \frac{\Delta l}{l_0} = \frac{\mathbf{x}_0^T \mathbf{u}}{l_0^2} \tag{7.24}$$

The advantage of this formulation to those that are formulated in local coordinates and then transformed into the global system is that the transformation is avoided. For a bar in 3D the entire computation consists of obtaining the unnormalized directional vector, $\mathbf{x}_0$, evaluating its length and evaluate the exterior product which gives a 3×3 matrix. From this matrix the element stiffness matrix may be obtained by simple assignment.

stiffness

```
Bar.stiffness():
  x = x0()
  xx = x * x
  size = no_dof * no_nodes
  Matrix Ke(size,size)
  for (i=1 to no_dof) do
    for (j=1 to no_dof) do
      Ke(i,j) = Ke(i+no_dof,j+no_dof) = xx(i,j)
      Ke(i+no_dof,j) = Ke(i,j+no_dof) = -xx(i,j)
  EA = material.put_par(1) * property.put_par(1)
  l3 = length*length*length
  factor = EA/l3
  return Ke*factor
```

The attribute, length, which stores the initial length of the bar, is introduced to make the stiffness method compatible with the linear stiffness in the non-linear bar element. The method uses an overloaded operator, Vector.operator *, to form the exterior product of two vectors, $\mathbf{x}\,\mathbf{x}^T$. A method x0 is defined for obtaining the directional vector from the node coordinates:

x0

```
Bar.x0():
   return node(2).put_coor() – node(1).put_coor();
```

The constitutive parameters, E and A, are defined by an Elastic object and a Property object. The Bar uses the Elastic class that was defined for the continuum elements, Section 5.1. The Property is a one parameter object of the same type as for the plane problems, Section 4.4.

The single strain component, the axial strain, is found from (7.24):

strain

```
Bar.strain():
   a = get_dof()
   Vector u(no_dof)
   for (i=1 to no_dof) do
      u(i) = a(i+no_dof) – a(i)
   x = x0()
   return dot(x,u)/(length*length)
```

where dot calculates the scalar product of 2 vectors. The conjugate stress - the axial force - is given by

$$N = EA\,\varepsilon \tag{7.25}$$

In pseudo-code this becomes

stress

```
Bar.stress():
   return material.put_par(1) * property.put_par(1) *strain()
```

The strain and the stress are constant within the element, thus it is only necessary to evaluate these in 1 point. This point may be referred to as a generalized Gauss point, therefore no_gauss = 1. The Bar element is presented in Figure 7.2.

7.3 Geometrically non-linear bar elements

The linear bar element presented above was a by-product of the non-linear formulation given in Section 7.1. A non-linear bar element, NlBar, which uses updated Lagrange formulation, (7.22), is derived from Bar.

The evaluation of the linear part of the stiffness can be carried out by Bar:stiffness by letting the method x0 be virtual. The updated geometry at a new equilibrium point is $\mathbf{x} = \mathbf{x}_0 + \mathbf{u}$:

x0

```
NlBar.x0():
   a = get_dof()
   Vector u(no_dof)
   for (i=1 to no_dof) do
      u(i) = a(i+no_dof) – a(i)
   return Bar.x0() + u
```

All scaling, however, refers to the initial length of the bar and it is therefore necessary to have the extra attribute, length, which is initialized with the original length of the bar, l_0. In this way Bar.stiffness represents exactly the linear part of the stiffness. The total stiffness is evaluated as the sum of the linear contribution obtained from Bar.stiffness and the initial stress term,

stiffness

```
NlBar.stiffness():
   Ke = Bar.stiffness()
   EA = material.put_par(1) * property.put_par(1)
   factor = EA*strain()/length
   for (i=1 to no_dof) do
      for (j=1 to no_dof) do
         Ke(i,j) += factor
         Ke(i+no_dof,j+no_dof) += factor
         Ke(i+no_dof,j) –= factor
         Ke(i,j+no_dof) –= factor
   return Ke
```

The initial stress part of the stiffness is evaluated on basis of the strain. In ObjectFEM the stiffness is always evaluated from the dof that are stored in the nodes. These usually refer to the previous equilibrium state corresponding to a modified Newton-Raphson update.

The internal force is evaluated from (7.11) using the current displacement state:

intforce

```
NlBar.intforce(a):
   Vector u(no_dof);
   for (i=1 to no_dof) do
      u(i) = a(i+no_dof) – a(i);
   Eg = green_strain(u)
   EA = material.put_par(1) * property.put_par(1)
   factor = Eg * EA / length
   p = factor * (Bar.x0()+u)
   Vector fint(no_dof*no_nodes)
   for (i=1 to no_dof) do
      fint(i) = –p(i)
      fint(i+no_dof) = p(i)
   return fint
```

The method, green_strain, which takes the current deformation, $\mathbf{u}$, as argument, calculates the corresponding strain using (7.5):

green strain

```
NlBar.green_strain(u)
   return dot(Bar.x0()+0.5*u,u)/(length*length);
```

The methods, intforce and green_strain, use the initial direction vector, $\mathbf{x}_0$. However, the directional vector evaluated by the method, x0, refers to the updated configuration. The initial configuration must instead be obtained from the method Bar.x0, that does not include the previous deformation of the bar. The definition of the virtual method, strain, concludes the implementation of NlBar. It uses get_dof to obtain the converged dof values from the nodes, i.e.

strain

```
NlBar.strain()
   a = get_dof()
   Vector u(no_dof)
   for (i=1 to no_dof) do
      u(i) = a(i+no_dof) – a(i)
   return green_strain(u)
```

Here it is assumed that the Nodes have stored the updated dof before the strain is evaluated by the Element. The stress is evaluated by the inherited method Bar.stress. The bar element class is defined in Figure 7.2.

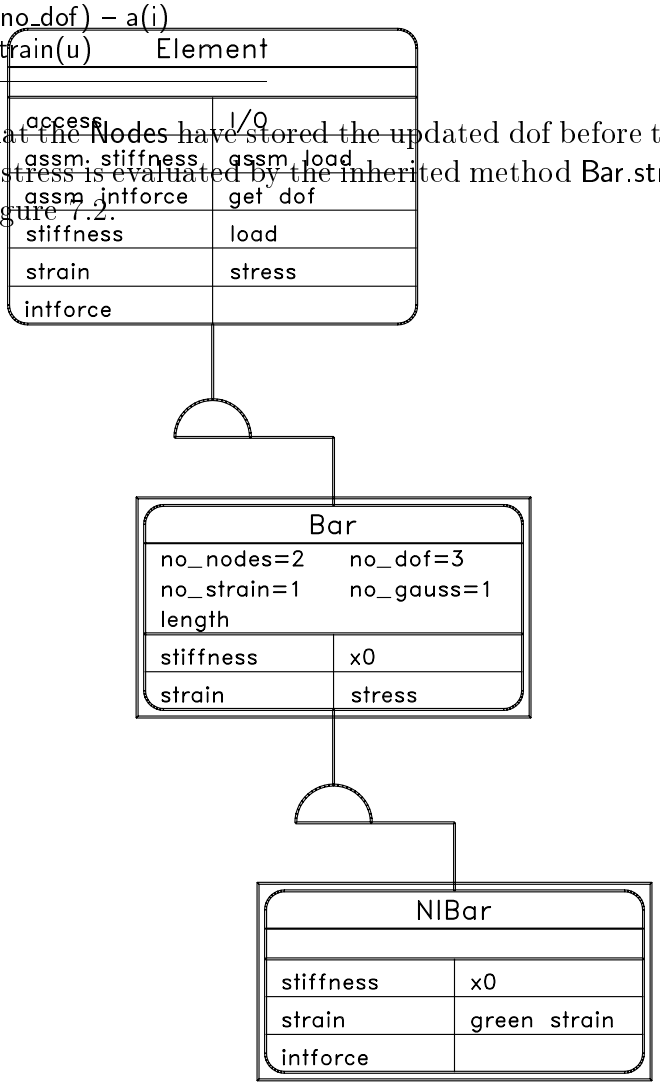


Figure 7.2: Linear elastic bar elements

7.4 Examples

In Chapter 6 two solution algorithms were formulated. The arc-length method of Crisfield (1981), Algorithm 6.1, and the orthogonal residual method, Algorithm 6.2. In the following two examples of non-linear truss structures are analyzed using the two algorithms. The intention is to demonstrate the behaviour of the algorithms at characteristic points, such as limit points in the load-displacement space and in the displacement space. The issue is not to compare the numerical efficiency of the methods - for such a purpose the examples are too small and too special.

7.4.1 Example 1: Two-bar truss

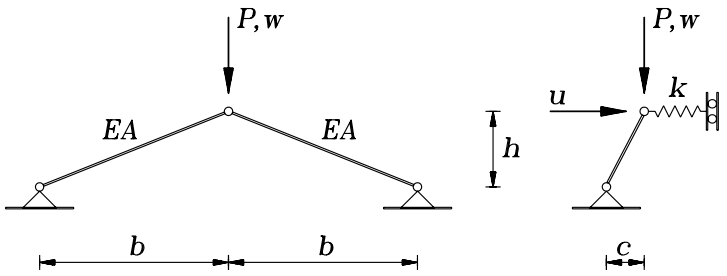


Figure 7.3: Two-bar truss with lateral support by a spring

A symmetric two-bar truss supported by a lateral spring is shown in Figure 7.3. The truss is tilted with an inclination, c, which could be regarded as a geometric imperfection of the ideal structure, $c = 0$. A vertical downward load, P, acts in the center node. The structure is fixed at the two end nodes, while the spring remains horizontal during deformation. The structure is described by two displacement components: the vertical displacement of the center node, w, and the horizontal displacement, u, in the direction of the spring. Due to symmetry the other horizontal displacement component is zero. The truss consists of two non-linear bar elements with axial stiffness, EA, and an initial length, l_0. The lateral support is provided by a linear spring with the stiffness, k.

The truss is described by the non-dimensional parameters: height $h/b = 0.2$, inclination with vertical, $c/h = 0.005$, and the spring stiffness $kb/EA = 0.02$. A suitable load increment, ΔP, may be estimated from the bifurcation load of the ideal vertical structure $P_b = 0.0027EA$, see e.g. Krenk (1993a). A load increment of $\Delta P = 0.001EA$ thus gives approximately 3–4 equilibrium points before the first limit point is reached. The maximum step length l_{max} is adjusted if the number of equilibrium iterations differs from a desired number, $i_d = 3$. For the arc-length method the continuous modification of the step length is used with $\alpha = 0.5$, whereas the discrete modification is used for the orthogonal residual method, cf. Section 6.2.2. In both algorithms the absolute maximum step length is set to twice the initial increment, i.e. $C = 2$ in (6.45). The algorithms are restarted with half the previous increment size if the number of equilibrium iterations exceeds $i_{max} = 6$. Convergence is measured by the reduced Euclidian norm with a threshold $\varepsilon = 10^{-3}$, (6.50).

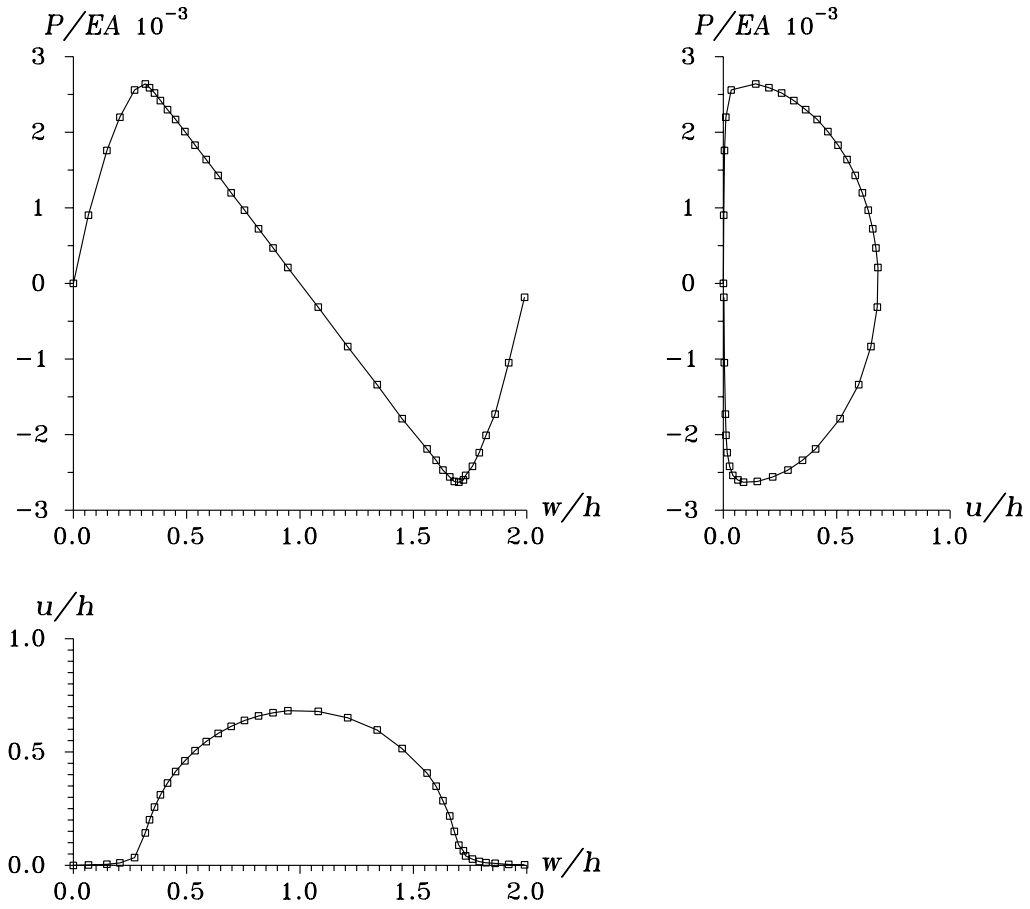


Figure 7.4: Equilibrium path for the two-bar truss

Traces of the equilibrium path evaluated by the orthogonal residual method are shown in Figure 7.4. It is seen that the equilibrium points are concentrated around the load limit points. At these points the direction in the displacement space changes dramatically and it is therefore necessary for the algorithm to restart the load step with a smaller increment. This reduced increment size is kept until the number of iterations becomes less than $i_d = 3$, i.e. at $P \approx 0$. The load-displacement curve $w - P$ is almost linear and therefore the step length could presumably be increased without loosing any information. However, this linearity hides a non-linearity in the $w - u$ space. The equal spacing on the equilibrium points on this linear part shows that the step length restriction is active thus the effective stiffness is less than $C^{-1} = 1/2$ of the initial stiffness.

The critical points in the analysis are the load limit points where the displacement increment must change its direction. Figure 7.5 illustrates how the orthogonal residual method and the arc-length method iterate around the first load limit point. In the orthogonal residual algorithm the displacement increment consists of a contribution along the current direction and a correction produced by the residual. The initial increment is evaluated from the tangent stiffness and does not represent the direction of the final increment. In the orthogonal residual method the magnitude of the initial increment is kept constant

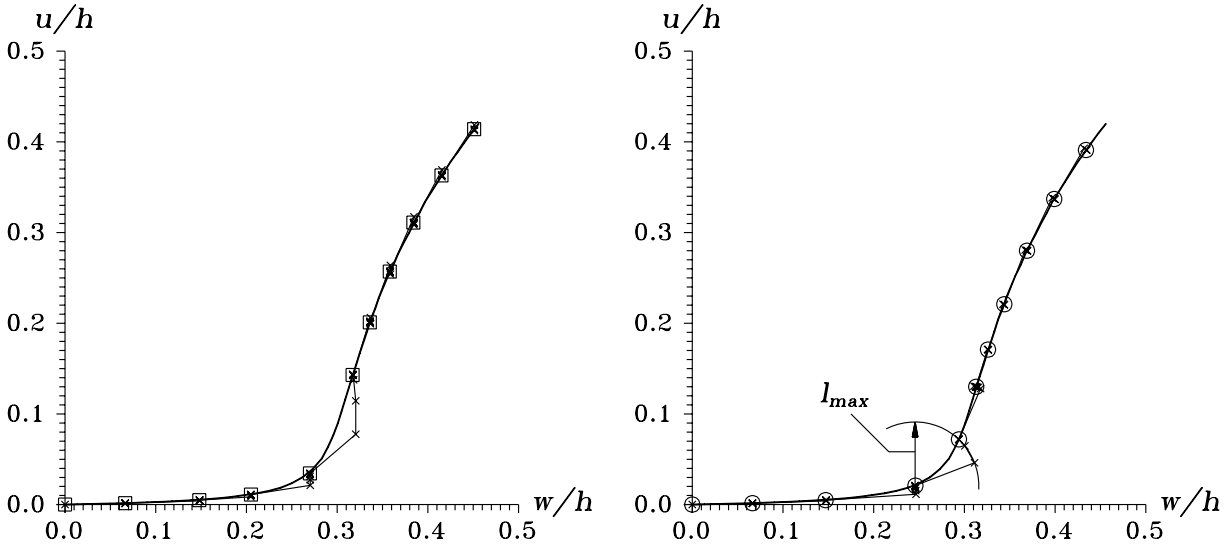


Figure 7.5: Equilibrium iterations at load limit point: □ orthogonal residual method, ⊙ arc-length method

and the subincrements give additional contributions. The final increment is thus allowed to exceed the maximum step length and therefore the total increment around the limit point becomes relatively large. The arc-length, however, the total increment must fulfil the constraint that the arc-length is constant for all increments. For the present version of the arc-length method the constraint forms a circle (hyper-sphere) in the displacement space on which the iterative increments are situated, cf. Figure 7.5.

Table 7.2: Two-bar truss analysis

	Orthogonal residual						Arc-length		
	Discrete			Continuous			Continuous		
$\Delta P/EA$	n_{tot}	i_{tot}	res	n_{tot}	i_{tot}	res	n_{tot}	i_{tot}	res
$0.5 \cdot 10^{-3}$	55	184	3	57	172	1	55	166	0
$0.75 \cdot 10^{-3}$	45	187	7	49	156	2	58	195	5
$1.0 \cdot 10^{-3}$	37	164	5	44	143	2	44	148	3
$1.5 \cdot 10^{-3}$	33	159	7	41	139	3	40	131	1

The truss is analyzed with different load increments. The orthogonal residual method is with both the discrete adaptive modification of the step length and the continuous modification as used for the arc-length method. Table 7.2 summarizes the number of load steps, n_{tot}, the total number of iterations, i_{tot}, and the number of restarts, res, used to compute the entire equilibrium path, Figure 7.4, for different load increments. The two methods that use the continuous schemes produce almost identical results. It is seen that the orthogonal residual method with discrete adjustment of the step length uses fewer load steps than both analyses with continuous modification of the step length. However, the

number of iterations and the need for restart is higher for the discrete scheme. Usually, the computational overhead of the equilibrium iterations in a modified Newton-Raphson scheme is small compared to the stiffness update. This might suggest that the discrete adaptive scheme is to be used, but general conclusions can not be made from this special problem.

7.4.2 Example 2: 12-bar truss

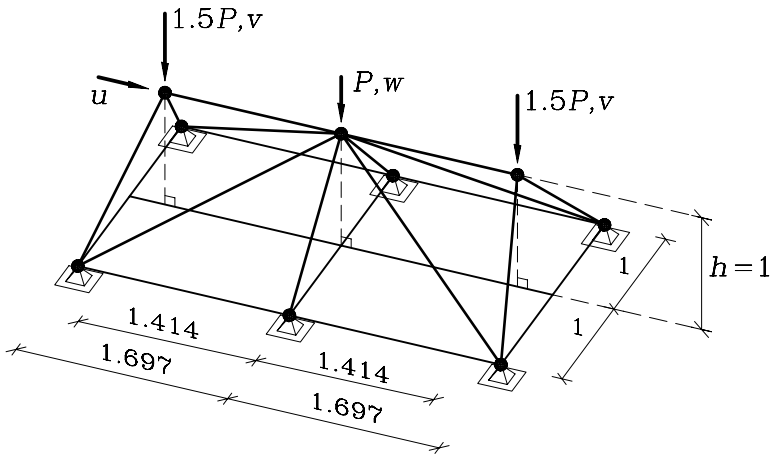


Figure 7.6: Space truss with 12 bars

A space truss, as shown in Figure 7.6, consists of 12 non-linear bar elements with axial stiffness, EA. Its dimensions shown in the figure are relative to the height, $h = 1$. The structure is fixed in 6 nodes which form the zero-plane. Three vertical forces act in the free nodes and have a magnitude of $1.5P$, P and $1.5P$, respectively. The symmetric deformation of the structure is described by two vertical components, v and w, and one horizontal component, u.

The complete deformation history resulting in an inverted form of the structure through a series of snap-throughs is illustrated by the nine states of Figure 7.7. In state 1 the side nodes snap through while the center node is pushed slightly upward. During this process the forces change from positive to negative. From state 2 the load again starts to increase pushing the center node down until it coincides with the zero-plane. At this point the load has decreased to zero. The load then becomes negative and the side nodes start to move upward until the completely plane structure, state 5, is reached. This is a state of complete symmetry on the load-displacement path, and the following states retraces the previous history in reverse order. Finally, yielding the inverted structure the all members come in tension and the structure shows a hardening behaviour.

The full equilibrium path for a 12 bar space truss has been traced using the arc-length method, Figure 7.8, and the orthogonal residual method, Figure 7.9. The load increment is to $\Delta P = 0.05EA$ whereby the first limit points is reached in 4 load steps. The maximum step length is adjusted if the number of iterations differ from $i_d = 3$. The arc-length

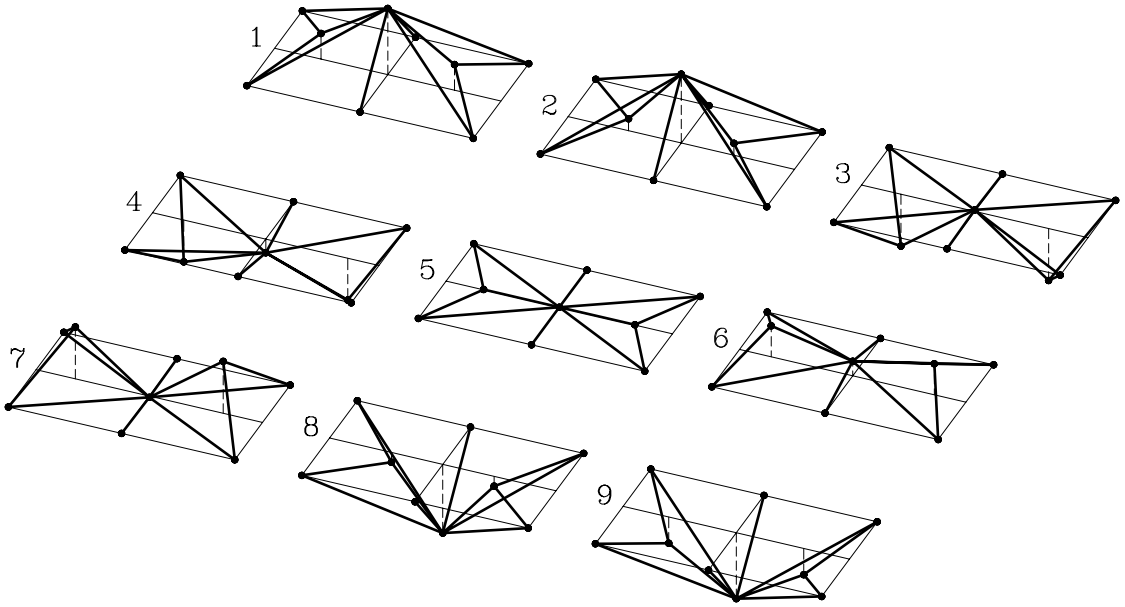


Figure 7.7: Deformed configurations of the 12-bar space truss

algorithm uses a continuous adaptive modification of the step length with $\alpha = 0.5$, while the orthogonal residual method uses the discrete adaptive scheme, cf. Section 6.2.2. In both algorithms the absolute maximum step length is set to twice the initial increment, i.e. $C = 2$ in (6.45). Both algorithms are restarted with half the previous increment size if the number of equilibrium iterations exceeds $i_{max} = 6$. Convergence is measured by the reduced Euclidian norm with a threshold $\varepsilon = 10^{-3}$, (6.50).

It is seen from the $w - v$ curve on Figure 7.8 that the arc-length method distributes the 95 equilibrium points uniformly. These points are obtained in 291 iterations without requiring any restarts. The displacement path is smooth and without dramatic changes in the direction, thus tracing the equilibrium path in the displacement space, as done by the arc-length method, is simple though not trivial because none of the displacement components increases monotonically.

The orthogonal residual method, Figure 7.9, concentrates the 100 equilibrium point in two regions: after state 3 and state 7. At these stationary points the displacement component, v, shows a excessive hardening behaviour. In regions where the stiffness increases rapidly the modified Newton-Raphson algorithm is known to have convergence problems. In the present example this problem is handled by using full Newton-Raphson stiffness updates during equilibrium iterations. The analysis require 8 restarts with reduced increments of which the 4 uses full Newton-Raphson updates. The full stiffness updates are used at state 3 and state 7, thus the concentration of equilibrium points are related to the requirement for full stiffness updates.

The full stiffness update is included by modifying the equilibrium iterations in solution algorithm, Algorithm 6.3. The design of the classes in ObjectFEM is aimed at modified Newton-Raphson stiffness updates, cf. Section 6.3. Still, the implementation of full Newton-Raphson updates can be done on application level by letting the nodes store displacements

that are not in equilibrium. The modified equilibrium iterations are given in Algorithm 7.1.

ALGORITHM 7.1: EQUILIBRIUM ITERATIONS WITH FULL NEWTON-RAPHSON UPDATES

```
 equilibrium iterations:
i = 0
do
{
  i += 1
  fi = 0
  for (elem=ellist.start() to ellist.end) do
    elem.assm_intforce(fi,a+da)
  dq = fi – fe
  if (OR = TRUE) xi = dot(dq,da)/dot(df,da)
  r = –dq + (xi * df)

  if (NR = TRUE)                    <-- extension
    K = 0
    for (elem=ellist.start() to ellist.end) do
      elem.assm_stiffness(K)
    factor(K,fix)

  solve(K,delta,r,fix)
  if (OR = TRUE and QN = TRUE)
    eta = –dot(dq,delta)/dot(dq,da)
    delta /= (1 + eta)
  else if (ARC = TRUE)
    dxi = dxi(da,da_start,delta,l_max)
    delta += dxi * da_start
    xi += dxi
  da += delta

  if (NR = TRUE)                    <-- extension
    for (node=nolist.start() to nolist.end) do
      node.get_disp(a+da)
}
until (rnorm(r)<EPS*rnorm(df) or i>i_max)
```

7.4.3 Concluding remarks

The version of the orthogonal residual algorithm used in these two examples uses a load controlled load incrementation strategy. This is not optimal because it leads to an unequally spaced distribution of the equilibrium points - as it is clearly seen on the equilibrium curves, e.g. Figure 7.9. Experience has shown that if the load incrementation is displacement controlled, the distribution of the equilibrium points becomes better. Therefore orthogonal residual algorithm should use displacement control, e.g. in the way that it is implemented in the elasto-plastic analysis, Section 8.5.

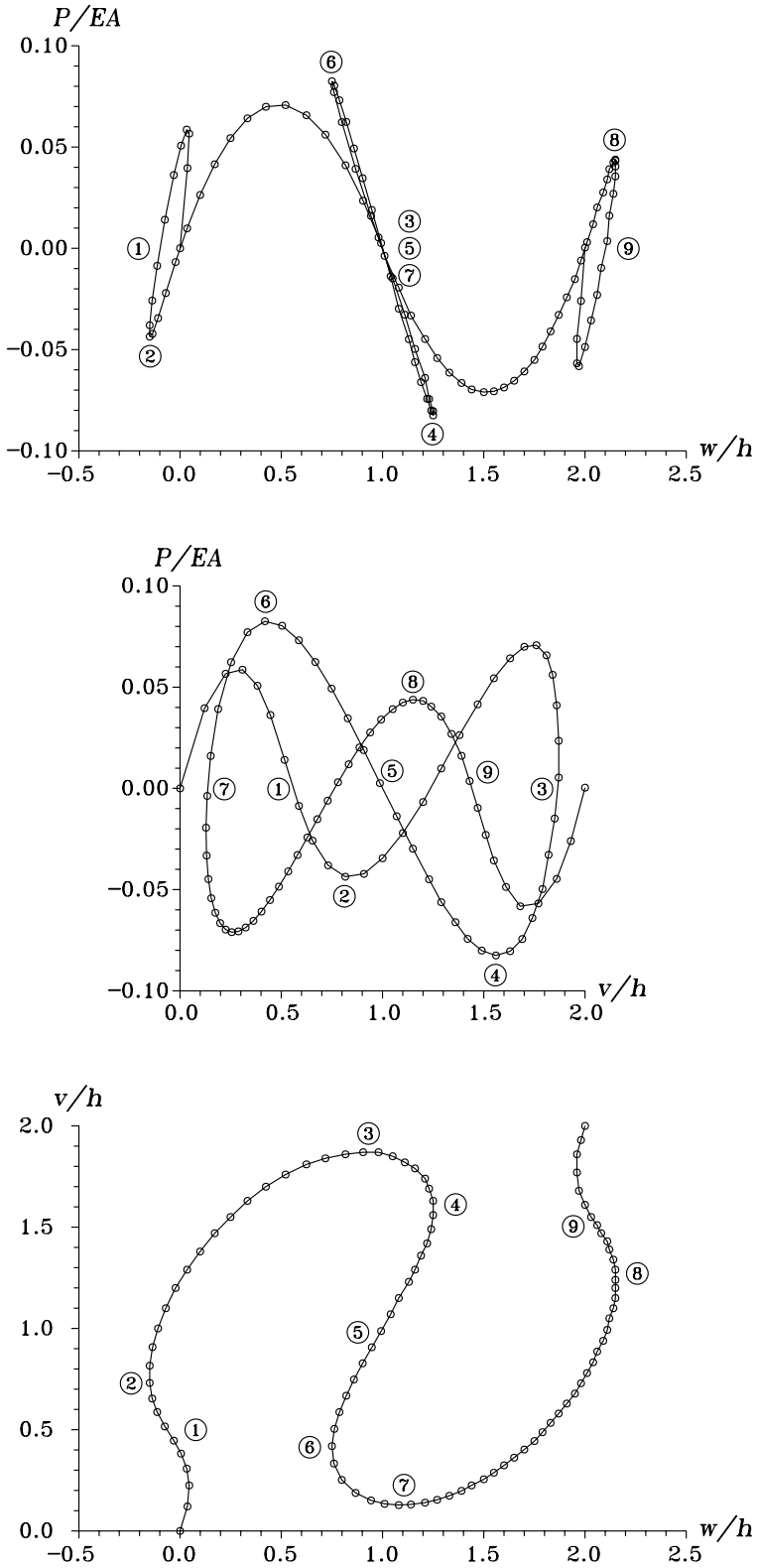


Figure 7.8: Equilibrium path for 12-bar truss computed with the arc-length method

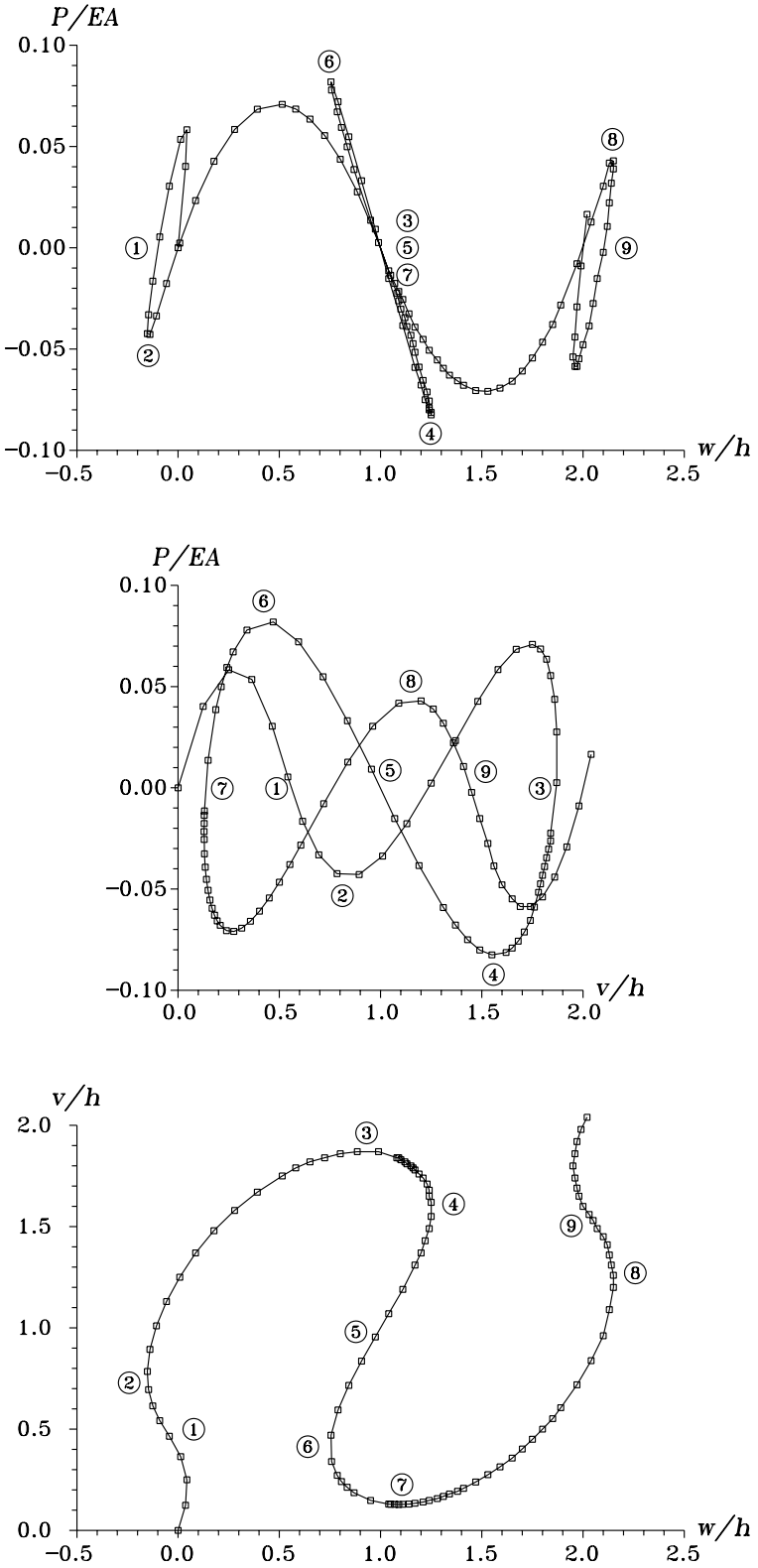


Figure 7.9: Equilibrium path for 12-bar truss computed with the orthogonal residual method

Chapter 8

Elasto-plastic materials

As described in Chapter 6 the non-linear problems can be divided in those where the internal force can be evaluated explicitly and those that are implicit, e.g. require integration over the entire load history. The elasto-plastic material model is an example of non-linear problems with implicit evaluation of the internal force. The global iteration schemes as presented in Chapter 6 are assumed to take care of the global equations. This chapter concentrates on the evaluation of the internal force, used in the global iterations, and the definition of a representative tangent stiffness.

The chapter shortly presents the theory for hardening plasticity introducing the incremental formulation that is to be integrated over the entire load history. From the constitutive relations it is possible to derive a tangent stiffness matrix based on the elasto-plastic constitutive matrix, $\mathbf{C}^{ep}$. Integration of the incremental constitutive relation is done numerically and different integration schemes are considered. It is assumed that Gauss integration is used, thus it is only necessary to consider a single material point within an element.

The additions to the FEM classes are mainly related to the integration of the constitutive relation. The internal force can be found using different strategies, still it is found that they fall into a common category that can be captured by few methods. These are used to define a Plastic material class, which inherits its elastic behaviour from the Elastic material class. A von Mises associated plasticity model is presented as an implementation of a elasto-plastic material model. The chapter is concluded by an example where the von Mises model is used in a convergence study of the orthogonal residual methods, cf. Chapter 6.

8.1 Hardening plasticity

Plasticity theory is based on an observation that a material in loading may experience plastic (irreversible) deformation and in a following unloading phase regains its elastic properties, see e.g. Chen & Han (1998). To describe the behaviour of an elasto-plastic material the constitutive model must add two things to the elastic relations: a definition of plastic loading and elastic unloading, and a flow rule describing the development of the plastic deformation.

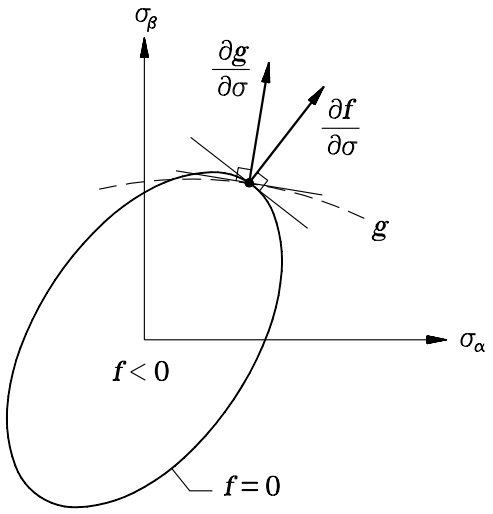


Figure 8.1: Yield function and plastic potential

In plasticity theory the current loading state of a material point is determined by the yield function, f, see e.g. Figure 8.1. The yield function is defined in terms of the stress state, $\boldsymbol{\sigma}$, and some state parameters, $\boldsymbol{\alpha}$, i.e.

$$f = f(\boldsymbol{\sigma}, \boldsymbol{\alpha}) \leq 0 \tag{8.1}$$

The stress, $\boldsymbol{\sigma}$, defines the position in the domain and the state parameters, $\boldsymbol{\alpha}$, describes the size, shape and position of the yield surface relative to an initial configuration where only elastic deformation has taken place. The yield function divides the stress space into two domains: an elastic domain where $f < 0$ and a plastic domain, $f = 0$, referred to as the yield surface.

A stress increment, $d\boldsymbol{\sigma}$, alters the state of a material point. If the material point is in the elastic domain, $f(\boldsymbol{\sigma}, \boldsymbol{\alpha}) < 0$, elastic deformations will occur. For a stress point on the yield surface, $f(\boldsymbol{\sigma}, \boldsymbol{\alpha}) = 0$, a stress increment can either give elastic unloading, thus leaving the yield surface and entering the elastic domain, or it might produce additional plastic deformation and the point remains on the yield surface. For hardening plasticity the unloading-loading condition follows from the Drucker postulate stating that the yield surface is convex, see e.g. Ottosen (1987),

$$\left(\frac{\partial f}{\partial \boldsymbol{\sigma}}\right)^T d\boldsymbol{\sigma} = \begin{cases} < 0 & \text{elastic unloading} \\ \geq 0 & \text{plastic loading} \end{cases} \tag{8.2}$$

Loading is thus determined by the sign of the projection of the stress increment, $d\boldsymbol{\sigma}$, onto the outward normal of the yield surface, see Figure 8.2.

The deformation in the elastic domain follows classical elasticity theory, e.g. Hooke's law. A flow rule is introduced to describe the deformation taking place during plastic loading. A stress increment, $d\boldsymbol{\sigma}$, leads to an increment in the conjugate strain, $d\boldsymbol{\varepsilon}$. The strain is divided in an elastic and a plastic part,

$$d\boldsymbol{\varepsilon} = d\boldsymbol{\varepsilon}^e + d\boldsymbol{\varepsilon}^p \tag{8.3}$$

The stress increment is common for both the elastic and the plastic strain thus can be found from the elastic part of the strain using the incremental constitutive relation,

$$d\boldsymbol{\sigma} = \mathbf{C}\, d\boldsymbol{\varepsilon}^e = \mathbf{C}\,(d\boldsymbol{\varepsilon} - d\boldsymbol{\varepsilon}^p) \tag{8.4}$$

where $\mathbf{C}$ is the elastic constitutive matrix, e.g. (5.7). The flow rule relates the development of plastic strain, $d\boldsymbol{\varepsilon}^p$, to a plastic potential, $g(\boldsymbol{\sigma}, \boldsymbol{\alpha})$, i.e.

$$d\boldsymbol{\varepsilon}^p = \frac{\partial g}{\partial \boldsymbol{\sigma}} d\lambda \tag{8.5}$$

where $d\lambda$ is the plastic multiplier. (8.5) states that the plastic part of the strain acts in the direction that is normal to the plastic potential with a magnitude scaled by $d\lambda$, see e.g. Figure 8.1. Plasticity theory where the yield function is used as plastic potential, $g = f$, is called associated plasticity. This gives that the plastic strain develops in a direction normal to the yield surface and (8.5) may therefore be referred to as a normality rule. Normality implies that if the yield function is independent of some stress component, e.g. the mean stress, then the plastic strain corresponding to this component, i.e. the dilatation, will be zero. This representation applies well to metal plasticity. For soil and granular materials the flow direction is usually not normal to the yield surface and a non-associated flow rule, $g \neq f$, must be used, Crisfield (1991).

In case of plastic loading consistency requires that the stress increment, $d\boldsymbol{\sigma}$, does not increase the value of the yield function. Thus the increment in the yield function must be 0, giving that

$$df = \left(\frac{\partial f}{\partial \boldsymbol{\sigma}}\right)^T d\boldsymbol{\sigma} \; + \; \left(\frac{\partial f}{\partial \boldsymbol{\alpha}}\right)^T d\boldsymbol{\alpha} = 0 \tag{8.6}$$

This means that the stress can only be increased if the shape or position of the yield surface changes. For work hardening materials the evolution of the yield surface depends on the development of plastic strain, see e.g. Chen & Han (1988). The increment in the state variables, $d\boldsymbol{\alpha}$, is related to the increment in the plastic strain, $d\boldsymbol{\varepsilon}^p$, through the plastic multiplier, $d\lambda$,

$$d\boldsymbol{\alpha} = \mathbf{h}\, d\lambda \tag{8.7}$$

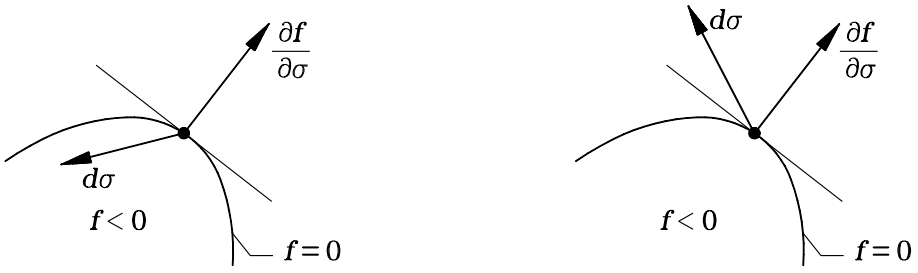


Figure 8.2: Hardening plasticity: a) unloading, b) plastic loading

where $\mathbf{h}(\boldsymbol{\sigma}, \boldsymbol{\alpha})$ is a set of hardening functions. Inserting this into the consistency relation, (8.6), yields

$$\left(\frac{\partial f}{\partial \boldsymbol{\sigma}}\right)^T d\boldsymbol{\sigma} \; - \; H \, d\lambda = 0 \tag{8.8}$$

with the hardening modulus, H, defined as

$$H = - \left(\frac{\partial f}{\partial \boldsymbol{\alpha}}\right)^T \mathbf{h} \tag{8.9}$$

Inserting (8.5) into the constitutive relation, (8.4), and combining it with (8.8) gives an expression for the plastic multiplier

$$d\lambda = \frac{\left(\mathbf{C} \dfrac{\partial f}{\partial \boldsymbol{\sigma}}\right)^T}{H + \left(\dfrac{\partial f}{\partial \boldsymbol{\sigma}}\right)^T \mathbf{C} \left(\dfrac{\partial g}{\partial \boldsymbol{\sigma}}\right)} \, d\boldsymbol{\varepsilon} \tag{8.10}$$

Using (8.10) and (8.5) in the incremental constitutive relation, (8.4), enables the stress increment to be stated in terms of the total strain increment,

$$d\boldsymbol{\sigma} = \left[\mathbf{C} - \frac{\left(\mathbf{C} \dfrac{\partial g}{\partial \boldsymbol{\sigma}}\right) \left(\mathbf{C} \dfrac{\partial f}{\partial \boldsymbol{\sigma}}\right)^T}{H + \left(\dfrac{\partial f}{\partial \boldsymbol{\sigma}}\right)^T \mathbf{C} \left(\dfrac{\partial g}{\partial \boldsymbol{\sigma}}\right)} \right] d\boldsymbol{\varepsilon} \tag{8.11}$$

which is a tangent stiffness relation on the form

$$d\boldsymbol{\sigma} = \mathbf{C}^{ep}(\boldsymbol{\sigma}, \boldsymbol{\alpha}) \, d\boldsymbol{\varepsilon} \tag{8.12}$$

$\mathbf{C}^{ep}$ is the elasto-plastic constitutive matrix. The first term is the pure elastic matrix and the last term is the correction due to development of plastic strain. It should be noted that the correction is only active if the material point is in the plastic domain. The elasto-plastic constitutive matrix, $\mathbf{C}^{ep}$, represents the tangent behaviour of the material point at a given stress state. This constitutive relation should thus be used to establish the tangent stiffness, (6.6), used in the next load step. It is noted that for non-associated plasticity the elasto-plastic matrix is unsymmetric, thus the tangent stiffness matrix, $\mathbf{K}_t$, also becomes unsymmetric.

Table 8.1: Concepts in plasticity theory

Strain:	$d\boldsymbol{\varepsilon} = d\boldsymbol{\varepsilon}^e + d\boldsymbol{\varepsilon}^p$
Stress:	$d\boldsymbol{\sigma}$
Yield function:	$f(\boldsymbol{\sigma}, \boldsymbol{\alpha})$
Plastic potential:	$g(\boldsymbol{\sigma}, \boldsymbol{\alpha})$
Flow rule:	$d\boldsymbol{\varepsilon}^p = d\lambda\,(\partial g/\partial \boldsymbol{\sigma})$
Evolution law:	$d\boldsymbol{\alpha} = d\lambda\,\mathbf{h}(\boldsymbol{\sigma}, \boldsymbol{\alpha})$
Constitutive model:	$d\boldsymbol{\sigma} = \mathbf{C}^{ep}\, d\boldsymbol{\varepsilon}$

8.1.1 Hardening rules

The hardening rules define the modification of the yield surface that takes place due to plastic flow. The size, shape and position of the yield surface is determined by the state variables, $\boldsymbol{\alpha}$, whose development is related to the plastic deformation through the plastic multiplier, $d\lambda$, cf. (8.7). A yield surface may be written on the following form

$$f(\boldsymbol{\sigma}, \boldsymbol{\alpha}) = F(\boldsymbol{\sigma} - \bar{\boldsymbol{\alpha}}) - \sigma_Y(\boldsymbol{\kappa}) = 0 \tag{8.13}$$

where the state parameters, $\boldsymbol{\alpha}$, are divided into two sets

$$\boldsymbol{\alpha} = \left\{ \begin{array}{c} \bar{\boldsymbol{\alpha}} \\ \boldsymbol{\kappa} \end{array} \right\} \tag{8.14}$$

The yield stress, $\sigma_Y(\boldsymbol{\kappa})$, determines the size of the yield surface. Its development is described through the parameters, $\boldsymbol{\kappa}$. The function, $F(\boldsymbol{\sigma} - \bar{\boldsymbol{\alpha}})$, describes the shape and position of the yield surface. The parameters, $\bar{\boldsymbol{\alpha}}$, represents a translation of the yield surface from its initial center. These parameters are sometimes termed pseudo-stress or back-stress due to their direct relation with the stress, $\boldsymbol{\sigma}$, see e.g. Figure 8.3b.

It is noted that the yield surface retains its shape, thus (8.13) can not describe arbitrary hardening. The formulation, however, captures two distinct types of hardening models: isotropic hardening and kinematic hardening. In an isotropic hardening model the yield surface expands in all directions, but retains the shape and position, thus the yield stress increases in all directions, Figure 8.3a. In kinematic hardening, Figure 8.3b, the yield surface translates, but retains its shape and size. These two simple hardening models can be used for problems without load reversals. For materials in cyclic loading the stress path after a load reversal is not only depending on the current state, but also the previous cycles, thus the model should include some kind of memory as well, Ristinmaa (1993).

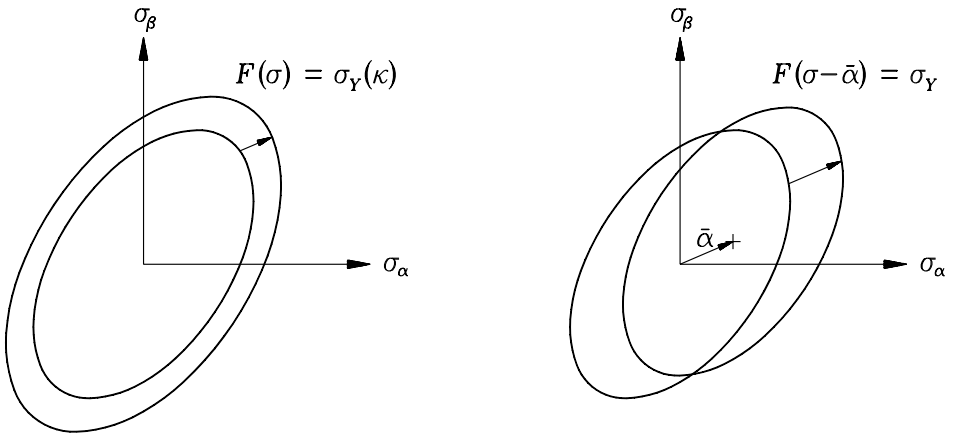


Figure 8.3: Hardening rules: a) isotropic, b) kinematic

8.2 Integration of stress

Non-linear solution strategies use the residual force to modify the current displacement estimate, $\mathbf{a}$. The residual is the difference between the external loads and the internal force corresponding to a displacement estimate. The internal force is found by integration of the total stress in each element, i.e.

$$\mathbf{f}_{int}^e = \int_{\Omega^e} \mathbf{B}^T \boldsymbol{\sigma}(\mathbf{a})\, d\Omega \tag{8.15}$$

In each point the total stress is calculated as a sum of two contributions: the stress at the previous equilibrium state, $\boldsymbol{\sigma}_0$, and an additional stress increment, $\Delta\boldsymbol{\sigma}$, i.e.

$$\boldsymbol{\sigma} = \boldsymbol{\sigma}_0 + \Delta\boldsymbol{\sigma} \tag{8.16}$$

The finite stress increment, $\Delta\boldsymbol{\sigma}$, in a point is found by integrating the infinitesimal constitutive relation, (8.12),

$$\Delta\boldsymbol{\sigma} = \int d\boldsymbol{\sigma} = \int_{\boldsymbol{\varepsilon}_0}^{\boldsymbol{\varepsilon}_0+\Delta\boldsymbol{\varepsilon}} \mathbf{C}^{ep}\, d\boldsymbol{\varepsilon} \tag{8.17}$$

The finite increment in the strain, $\Delta\boldsymbol{\varepsilon}$, is found from the estimate on displacement increment that is provided by the global solution algorithm, i.e.

$$\Delta\boldsymbol{\varepsilon} = \mathbf{B}\,\Delta\mathbf{a} \tag{8.18}$$

The integral, (8.17), must be evaluated numerically. There are two sources of integration error: a global discretization error and a local discretization error. The global solution algorithm provides finite strain increments, $\Delta\boldsymbol{\varepsilon}$. These represent a piecewise linear approximation to the true strain path. The total stress used in (8.15) therefore represents an approximation to the true equilibrium state, even if the integral, (8.17), in the previous load steps has been evaluated exactly. This type of integration error can only be reduced

by letting the increment size be relatively small. The local discretization error is related to the approximation of the integration path. The path followed in the integration is dependent on the current state, $\boldsymbol{\sigma}$ and $\boldsymbol{\alpha}$, and because the relation is non-linear it is generally necessary to use a iterative scheme. The following sections give a brief introduction to integration of the incremental constitutive relations identifying the standard parts of the integration schemes. A more comprehensive description is given e.g. in Crisfield (1991).

8.2.1 Explicit integration

In the forward Euler scheme the infinitesimal relation is replaced by a finite incremental relation,

$$\Delta\boldsymbol{\sigma} = \mathbf{C}^{ep}(\boldsymbol{\sigma}_0, \boldsymbol{\alpha}_0)\,\Delta\boldsymbol{\varepsilon} \tag{8.19}$$

where the tangent stiffness, $\mathbf{C}^{ep}$, is evaluated at the previous equilibrium point. A similar scheme can be employed for the update of the state parameters, $\boldsymbol{\alpha}$. This linear approximation can be refined to a piecewise linear integration using the subincremental method, see e.g. Crisfield (1991). In the subincremental methods the strain increment, $\Delta\boldsymbol{\varepsilon}$, is divided into m subincrements, i.e.

$$\delta\boldsymbol{\varepsilon} = \frac{\Delta\boldsymbol{\varepsilon}}{m} \tag{8.20}$$

The finite stress increment is then determined as the sum of m stress subincrements, $\delta\boldsymbol{\sigma}_k$, each evaluated as a forward Euler step:

$$\delta\boldsymbol{\sigma}_k = \mathbf{C}^{ep}(\boldsymbol{\sigma}_{k-1}, \boldsymbol{\alpha}_{k-1})\,\delta\boldsymbol{\varepsilon} \quad , \quad \Delta\boldsymbol{\sigma}_{k-1} = \sum_{i=1}^{k-1} \delta\boldsymbol{\sigma}_i \tag{8.21}$$

where the updated values of the stress, $\boldsymbol{\sigma}_{k-1}$, and the state parameter, $\boldsymbol{\alpha}_{k-1}$, are used in the evaluation of the tangent stiffness. Another simple refinement of the forward Euler scheme is a two-step procedure presented by Zienkiewicz & Taylor (1991). In the two-step procedure a forward Euler step is taken with half the total strain increment, i.e.

$$\Delta\boldsymbol{\sigma}_{1/2} = \mathbf{C}^{ep}(\boldsymbol{\sigma}_0, \boldsymbol{\alpha}_0)\,\frac{\Delta\boldsymbol{\varepsilon}}{2} \tag{8.22}$$

This mid-point state is assumed to be representative for the tangent stiffness and the full stress increment is evaluated as

$$\Delta\boldsymbol{\sigma} = \mathbf{C}^{ep}(\boldsymbol{\sigma}_{1/2}, \boldsymbol{\alpha}_{1/2})\,\Delta\boldsymbol{\varepsilon} \tag{8.23}$$

The methods presented above are referred to as explicit because the integration does not use any correction to ensure that the yield condition, (8.1), is fulfilled. Generally, the stress increments evaluated with these methods tend to drift away from the yield surface, see Figure 8.4, and the explicit schemes thus introduce errors that accumulate for each load step.

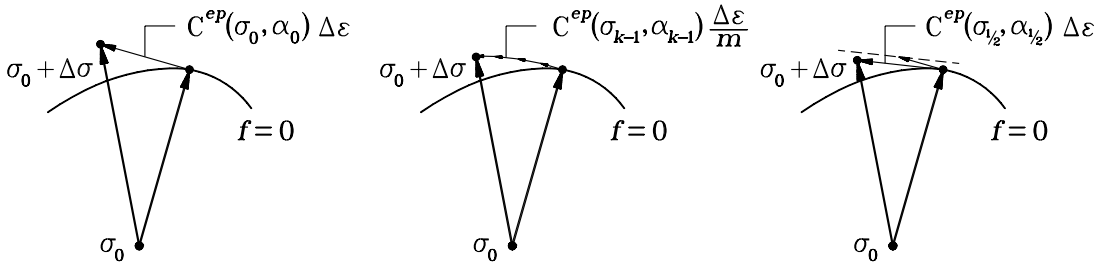


Figure 8.4: Explicit integration: a) forward Euler, b) subincremental method, c) mid-point method

8.2.2 Return mapping algorithms

The explicit integration schemes have the inherent problem that the evaluated stress does not lie on the yield surface, Figure 8.4. In the return mapping algorithms iterations are used to ensure that the final stress fulfils the yield condition, (8.1). The iterations are continued as long as the absolute value of the yield function exceeds a given tolerance limit. The method presented in the following is a one-step procedure, like the forward Euler scheme. Still, it can be refined with subincrements to increase the accuracy of the integration, see e.g. Krieg & Krieg (1977).

The stress increment, $\Delta\boldsymbol{\sigma}$, can be evaluated from the constitutive relation,

$$\Delta\boldsymbol{\sigma} = \mathbf{C}(\Delta\boldsymbol{\varepsilon} - \Delta\boldsymbol{\varepsilon}^p) \tag{8.24}$$

The total strain increment, $\Delta\boldsymbol{\varepsilon}$, is given, whereas the plastic strain, $\Delta\boldsymbol{\varepsilon}^p$, must be determined from (8.5). The direction of the plastic strain increment is determined by the gradient plastic potential, $\partial g/\partial\boldsymbol{\sigma}$. In the backward Euler scheme the stress increment, $\Delta\boldsymbol{\sigma}$, is

$$\Delta\boldsymbol{\sigma} = \mathbf{C}(\Delta\boldsymbol{\varepsilon} - \Delta\boldsymbol{\varepsilon}^p) = \mathbf{C}\Delta\boldsymbol{\varepsilon} - \Delta\lambda\mathbf{C}\frac{\partial g(\boldsymbol{\sigma}_n, \boldsymbol{\alpha}_n)}{\partial\boldsymbol{\sigma}} \tag{8.25}$$

where the gradient of the potential, $\partial g/\partial\boldsymbol{\sigma}$, is evaluated with the updated state, $\boldsymbol{\sigma}_n = \boldsymbol{\sigma}_0 + \Delta\boldsymbol{\sigma}$ and $\boldsymbol{\alpha}_n = \boldsymbol{\alpha}_0 + \Delta\boldsymbol{\alpha}$. The plastic multiplier, $\Delta\lambda$, is evaluated such that the yield condition is satisfied in the new point,

$$f(\boldsymbol{\sigma}_n, \boldsymbol{\alpha}_n) = 0 \tag{8.26}$$

In general the non-linear equations, (8.25) and (8.26), will not be satisfied by the first estimate. In the return mapping algorithms the stress increment is decomposed in an elastic predictor, $\Delta\boldsymbol{\sigma}^e$, and a plastic corrector, $\Delta\boldsymbol{\sigma}^p$,

$$\Delta\boldsymbol{\sigma} = \mathbf{C}\ (\Delta\boldsymbol{\varepsilon} - \Delta\boldsymbol{\varepsilon}^p) = \Delta\boldsymbol{\sigma}^e - \Delta\boldsymbol{\sigma}^p \tag{8.27}$$

where $\mathbf{C}$ is the elastic constitutive matrix. The elastic predictor is used to determine a first estimate on the stress, i.e.

$$\boldsymbol{\sigma} = \boldsymbol{\sigma}_0 + \Delta\boldsymbol{\sigma}^e \tag{8.28}$$

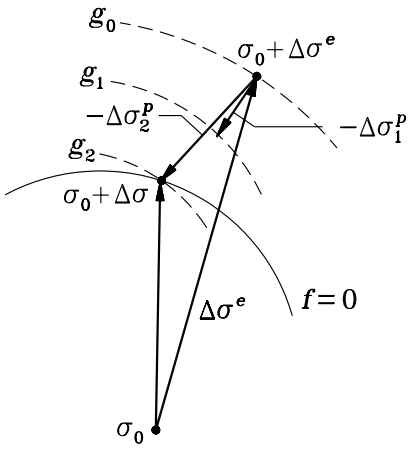


Figure 8.5: Backward Euler scheme: Return mapping algorithm

If this stress point remains in the elastic domain the stress is simply updated by the elastic term. If the stress prediction refers to a point outside the yield surface, $f(\boldsymbol{\sigma}_0+\Delta\boldsymbol{\sigma}^e, \boldsymbol{\alpha}_0) > 0$, the plastic correction must be evaluated. The plastic part of the strain is evaluated from the finite version of the flow rule, (8.5), giving the following correction,

$$\Delta\boldsymbol{\sigma}^p = \mathbf{C}\,\Delta\boldsymbol{\varepsilon}^p = \Delta\lambda\,\mathbf{C}\,\frac{\partial g}{\partial\boldsymbol{\sigma}} \tag{8.29}$$

where the plastic multiplier, $\Delta\lambda$, is evaluated from (8.10). The true direction of the plastic correction is determined by the gradient of the plastic potential in the final stress point which lies on the yield surface. However, this point is initially unknown and iterations are performed using the elastic predictor, $\boldsymbol{\sigma}_0+\Delta\boldsymbol{\sigma}^e$, as starting point. The following iterations then use the updated stress and state parameters to evaluate a new direction and plastic multiplier. Still, the correction refers to the elastic prediction, $\boldsymbol{\sigma}_0 + \Delta\boldsymbol{\sigma}^e$. The iteration process is illustrated in Figure 8.5 and the algorithm is summarized in Algorithm 8.1.

8.3 Classes in elasto-plastic analysis

The elasto-plastic equations presented in the previous sections refer to a single material point. The task of the Element is to collect and process these distributed properties and present them to the global solution algorithm. In elasto-plastic analysis the state of the material is individual for each point in the element, thus it is necessary to store the current state for each Gausspoint. However, the evolution of the stress and strain depends on the material model. Therefore the Material class needs to be modified to be able to evaluate the constitutive matrix, $\mathbf{C}$, and the increments in the stress, $\Delta\boldsymbol{\sigma}$, and state variables, $\Delta\boldsymbol{\alpha}$, for the individual Gausspoints. The Plastic subclass is introduced to define an elasto-plastic framework in which specific material models can be implemented.

ALGORITHM 8.1: RETURN MAPPING ALGORITHM

$$
\begin{aligned}
&\Delta\boldsymbol{\sigma}^e \;=\; \mathbf{C}\,\Delta\boldsymbol{\varepsilon}\\
&\texttt{if}\;\; (f(\boldsymbol{\sigma}_0+\Delta\boldsymbol{\sigma}^e,\boldsymbol{\alpha}) \;>\; 0)\\
&\quad \Delta\boldsymbol{\sigma} \;=\; \Delta\boldsymbol{\sigma}^e\\
&\quad \Delta\boldsymbol{\alpha} \;=\; \mathbf{0}\\
&\quad \texttt{do}\\
&\qquad H \;=\; -\left(\frac{\partial f(\boldsymbol{\sigma}_0+\Delta\boldsymbol{\sigma},\boldsymbol{\alpha}_0+\Delta\boldsymbol{\alpha})}{\partial\boldsymbol{\alpha}}\right)^T \mathbf{h}(\boldsymbol{\sigma}_0+\Delta\boldsymbol{\sigma},\boldsymbol{\alpha}_0+\Delta\boldsymbol{\alpha})\\
&\qquad \mathbf{n} \;=\; \frac{\partial f(\boldsymbol{\sigma}_0+\Delta\boldsymbol{\sigma},\boldsymbol{\alpha}_0+\Delta\boldsymbol{\alpha})}{\partial\boldsymbol{\sigma}}\\
&\qquad \tilde{\boldsymbol{\sigma}}^p \;=\; \mathbf{C}\,\frac{\partial g(\boldsymbol{\sigma}_0+\Delta\boldsymbol{\sigma},\boldsymbol{\alpha}_0+\Delta\boldsymbol{\alpha})}{\partial\boldsymbol{\sigma}}\\
&\qquad \Delta\lambda \;=\; \frac{\mathbf{n}^T\Delta\boldsymbol{\sigma}^e}{H+\mathbf{n}^T\tilde{\boldsymbol{\sigma}}^p}\\
&\qquad \Delta\boldsymbol{\sigma} \;=\; \Delta\boldsymbol{\sigma}^e \;-\; \Delta\lambda\,\tilde{\boldsymbol{\sigma}}^p\\
&\qquad \Delta\boldsymbol{\alpha} \;=\; \Delta\lambda\;\mathbf{h}(\boldsymbol{\sigma}_0+\Delta\boldsymbol{\sigma},\boldsymbol{\alpha}_0+\Delta\boldsymbol{\alpha})\\
&\quad \texttt{until}\;\; (|f(\boldsymbol{\sigma}_0+\Delta\boldsymbol{\sigma},\boldsymbol{\alpha}_0+\Delta\boldsymbol{\alpha})| \;<\; TOL)
\end{aligned}
$$

8.3.1 Extension to the Element class

The Element must supply two things in order for the solution algorithms to solve a non-linear problem: a tangent stiffness and the internal force.

The tangent stiffness does for elasto-plastic problems involve the elasto-plastic constitutive matrix, $\mathbf{C}^{ep}$. This matrix is dependent of the current loading state of the point and is therefore not constant over the element, as assumed in the evaluation of the stiffness matrix presented in Chapter 4. The evaluation of the constitutive matrix, $\mathbf{C}$, is therefore individual for each Gauss point. The modified stiffness computation becomes,

stiffness

```
Continuum.stiffness():
  for (i=1 to no_gauss) do
    dn = dN(i)
    jacobi = J(dn)
    b = B(dn,jacobi)
    dv = dV(i,jacobi)
    Ke += b.T * material.C(gausspoint(i)) * b * dv
  return Ke
```

It is important to realize that it is the responsibility of the Material class to evaluate the constitutive matrix, but the Gausspoint that stores the current state. The method, Material.C therefore takes a Gausspoint as argument.

The Element class is extended in connection with the development of non-linear solu-

tion methods, cf. Section 6.3. This added two methods: assm_intforce, which is a static method for extracting the displacement estimate and assembling the resulting internal force into the global force vector. The method, intforce, which is a virtual method called by assm_intforce, computes the internal force. The internal force in an element is evaluated from the displacement using the weighted integral of the stress, (6.1).

intforce

```
Continuum.intforce(a):
   Vector fint(no_dof*no_nodes)
   for (i=1 to no_gauss) do
      dn = dN(i)
      jacobi = J(dn)
      b = B(dn,jacobi)
      e = b * a;
      s = gausspoint(i).put_stress(e)
      fint += b.T * s * dV(i,jacobi)
   return fint
```

For elasto-plastic problems the computation requires the Material and Gausspoint classes to provide a number of methods such that the stress can be determined in any Gausspoint and at any displacement state. It is, however, noticed that the method, intforce, does not distinguish between linear and non-linear problems, relying only on the existence of the method, Gausspoint.put_stress. This makes it possible to mix linear and non-linear elements and material models in the same analysis.

8.3.2 The Gausspoint class

In path-independent problem such as continuum mechanics with finite deformation and non-linear elasticity theory, the Element can evaluate the internal force directly from the global strain estimate, thus there is no need for a local memory of the stress and strain history. In elasto-plastic problems the total stress needs to be integrated over the complete load history and it is necessary to save history information, i.e. the total strain, the total stress and the state parameters at the previous equilibrium point. These are individual for each material point and they are therefore stored in each integration point. The Gausspoint class have attributes to store the strain, stress and state variables, i.e. stress, strain and state. During the equilibrium iterations the internal force is evaluated in a number of intermediate states, each corresponding to increments in the stress, strain and state parameters. Upon convergence the total values are updated by their increments, therefore it is necessary to have three attributes that stores the increments during the equilibrium iterations, i.e. dstress, dstrain and dstate. To each of these attributes corresponds a set of simple access methods (set/put), see Figure 8.6. The method, Element.intforce, uses a special version of the Gausspoint.put_stress method, namely one taking the current total strain as input,

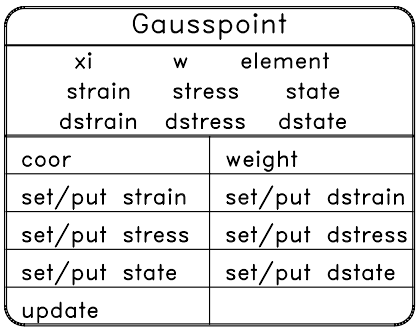


Figure 8.6: Extension of Gausspoint class

put stress

```
Gausspoint.put_stress(e):
   mat = element.material
   dstrain = e – strain
   dstress = mat.stress_inc(this)
   return stress+dstress
```

This method first evaluates the strain increment and from that the current increment in the stress and state parameters are obtained from Material.stress_inc. Notice that the reference to the material is obtained through the element attribute. The current state of the Gausspoint is obtained using an access method without any argument, e.g.

put stress

```
Gausspoint.put_stress():
   return stress+dstress
```

The strain, stress and state attributes are only updated explicitly when the new equilibrium point is detected. For this purpose an update method is introduced:

update

```
Gausspoint.update():
   strain += dstrain
   stress += dstress
   state += dstate
```

However, as it is only the Element that is explicitly called from the solution algorithm, the Element must have a method that communicates the message to each Gausspoint, i.e.

update

```
Element.update():
   for (i=1 to no_gauss) do
      gausspoint(i).update()
```

The extended Gausspoint class is shown in Figure 8.6.

8.3.3 The Plastic material class

The state of a particular integration point is stored in the Gausspoints. Still, the computation of the material properties is handled by the Material class. In elasto-plastic analysis the Material must be capable of providing a tangent stiffness for each Gausspoint and integrating the stress and state parameters for a given strain increment. To perform these tasks a general Material class for elasto-plastic analysis, Plastic, must define a yield function, gradients of the yield function and the plastic potential and methods for describing the hardening behaviour.

The elasto-plastic constitutive matrix, $\mathbf{C}^{ep}$, consists of an elastic part supplemented by the plastic correction. The elastic part is inherited from the Elastic material class, Section 5.3. In this way only plastic part of the computation lies in the hand of the material class Plastic. The modified constitutive method, C, takes the current Gausspoint as argument, so that it is possible to retrieve relevant information.

C

```
Plastic.C(gp):
   c = Elastic.C(gp)
   s = gp.put_stress()
   a = gp.put_state()
   if (yield(s,a)) < 0)
      return c
   n = dyield(s,a)
   se = c * n
   sp = c * dplast(s,a)
   h = H(s,a)
   denom = h + dot(n,sp)
   return (c – (sp * se))/ denom)
```

The method first retrieves the elastic part from Elastic. If the material point is elastic, the pure elastic part is returned. If there is plastic loading the plastic correction is calculated by use of (8.11). The method uses four additional methods: yield, dyield, dplast and H. As these are specific for each material model they must be defined as virtual methods of the Plastic class.

The integration of the stress increment is handled by the stress_inc method. As this method is used by the Gausspoint to evaluate the current state of the point, it is necessary to add this as a virtual method to the base class, Material. The implementation is provided by the subclass Plastic. In the present framework the return mapping scheme, Algorithm 8.1, is used for integrating the stress, i.e.

stress inc

```
Plastic.stress_inc(gp):
  c = Elastic.C(gp)
  dse = c * gp.put_dstrain()
  s0 = gp.put_stress()
  a0 = gp.put_state()
  if (yield(s0+dse,a0) < 0)
    return dse
  ds = dse
  da = 0
  do
    h = H(s0+ds, a0+da)
    n = dyield(s0+ds, a0+da)
    sp = c * dplast(s0+ds, a0+da)
    dl = dot(n,dse)/(h+dot(n,sp))
    ds = dse – dl * sp
    da = dl * hard(s0+ds, a0+da)
  until (abs(yield(s0+ds, a0+da)) < TOL)
  gp.set_dstate() = da
  return ds
```

The evolution of the state parameters, $\boldsymbol{\alpha}$, depends on a set of hardening functions, $\mathbf{h}$. These are needed during iterations and this adds another virtual method, hard, to the Plastic class.

The hardening modulus, H, is evaluated from the derivative of the yield function with respect to the state parameters, $\partial f/\partial \boldsymbol{\alpha}$, and the hardening functions, $\mathbf{h}$. The hardening modulus is thus common for all types of elastic problems and may therefore be defined as a static method in the Plastic class. The method takes the current stress and state vectors as input,

H

```
Plastic.H(s,a):
  return –dot(dyield_da(s,a),hard(s,a))
```

The method, dyield_da, defines the derivative of the yield function with respect to $\boldsymbol{\alpha}$. The Plastic class is presented in Figure 8.8.

8.4 von Mises plasticity

The Plastic class defines the elasto-plastic framework. Still, the virtual methods need to be interpreted in terms of an actual material model. This section introduces von Mises plasticity with linear isotropic hardening for three-dimensional continuum elements, cf. e.g. Chapter 5. The material model is simple but it contains all the features of a more advanced plastic material model.

In metal plasticity the plastic deformation is often assumed to occur without volume change, thus is independent of the mean stress. The mean stress can therefore be extracted from the stress measure used to describe the yield surface and the plastic flow. The stress

is decomposed in the mean stress,

$$\sigma_m = \frac{1}{3}\left(\sigma_{11} + \sigma_{22} + \sigma_{33}\right) \tag{8.30}$$

and the deviatoric stress components,

$$\sigma'_{\alpha\beta} = \sigma_{\alpha\beta} - \delta_{\alpha\beta}\sigma_m \quad , \quad \delta_{\alpha\beta} = \begin{cases} 1 & \text{if } \alpha = \beta \\ 0 & \text{otherwise} \end{cases} \tag{8.31}$$

This decomposition defines a stress space described by the hydrostatic axis, σ_m, and the deviatoric plane, $\boldsymbol{\sigma}'$, which is orthogonal to the hydrostatic axis.

von Mises used an associated flow rule to describe metal plasticity. The yield criterion is independent of the mean stress and can thus be formulated in terms of the deviatoric stress. It can be formulated in terms of the equivalent stress, σ_e, defined from the deviatoric stress, $\boldsymbol{\sigma}'$, i.e.

$$\sigma_e^2 = \frac{2}{3}\left(\sigma_1'\sigma_1' + \sigma_2'\sigma_2' + \sigma_3'\sigma_3'\right) \tag{8.32}$$

where σ_α' are the principal deviatoric stress components. The equivalent stress can be restated in the full stress components by use of (8.30) and (8.31),

$$\sigma_e^2 = \frac{1}{2}\left((\sigma_{11} - \sigma_{22})^2 + (\sigma_{22} - \sigma_{33})^2 + (\sigma_{33} - \sigma_{11})^2\right) + 3\left(\sigma_{12}^2 + \sigma_{23}^2 + \sigma_{31}^2\right) \tag{8.33}$$

The von Mises yield criterion with a linear isotropic hardening rule, see Section 8.1.1, can then be written as,

$$f(\boldsymbol{\sigma}, \boldsymbol{\alpha}) = \sigma_e(\boldsymbol{\sigma}') - (\sigma_Y + \alpha H) \tag{8.34}$$

The von Mises yield surface is a straight horizontal line in the meridian plane, (σ_m, σ_e), and forms a circle in the deviatoric stress plane with a radius equal to the current yield stress, $\sigma_Y + \alpha H$, Figure 8.7.

A formal evaluation of the hardening modulus, H, requires the single hardening function, h, to be defined. Having prescribed the linear hardening model the hardening function is determined by use of (8.9), i.e.

$$H = -\frac{\partial f}{\partial \alpha}\, h \quad \Rightarrow \quad h = 1 \tag{8.35}$$

The evolution of the state parameter, α, thereby becomes

$$\Delta\alpha = \Delta\lambda \tag{8.36}$$

Thus, the size of the yield surface follows directly from the plastic multiplier, $\Delta\lambda$.

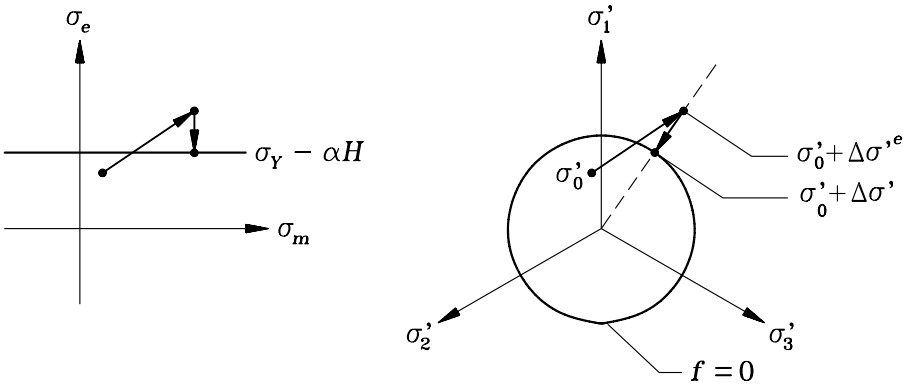


Figure 8.7: Radial return for von Mises plasticity: a) meridian plane, b) deviatoric plane

The yield stress, σ_Y, is independent of the stress. The gradient of the yield function can therefore be written as

$$\frac{\partial f}{\partial \boldsymbol{\sigma}} = \frac{\partial \sigma_e}{\partial \boldsymbol{\sigma}} = \frac{1}{2\sigma_e}\frac{\partial \sigma_e^2}{\partial \boldsymbol{\sigma}} \tag{8.37}$$

The return mapping algorithm, Algorithm 8.1, can be used to determine the stress increment, $\Delta\boldsymbol{\sigma}$. An elastic predictor, $\boldsymbol{\sigma}_0 + \Delta\boldsymbol{\sigma}^e$, is evaluated. The plastic correction, $\Delta\boldsymbol{\sigma}^p$, is then calculated in the updated point. This returns the stress along the gradient - radially - towards the yield surface, see Figure 8.7. As the yield surfaces in the two points are concentric circles the gradient is the same in the two points, the direction of the plastic correction is thus constant and only the magnitude, $\Delta\lambda$, which depends on the hardening modulus, H, remains to be determined. For linear hardening H is constant, thus the consistent point can be evaluated explicitly. Therefore the iterative strategy, Algorithm 8.1, can be replaced by an explicit radial return algorithm, which uses a Taylor expansion around the elastic predictor to determine the magnitude of plastic correction, see e.g. Krieg & Krieg (1977) and Crisfield (1991). For associated von Mises plasticity with linear isotropic hardening this becomes

$$\Delta\lambda \;=\; \frac{f(\boldsymbol{\sigma}_0 + \Delta\boldsymbol{\sigma}^e, \boldsymbol{\alpha}_0)}{H + \left(\dfrac{\partial f}{\partial \boldsymbol{\sigma}}\right)^T \mathbf{C} \left(\dfrac{\partial f}{\partial \boldsymbol{\sigma}}\right)} \;=\; \frac{f(\boldsymbol{\sigma}_0 + \Delta\boldsymbol{\sigma}^e, \boldsymbol{\alpha}_0)}{H + 3\mu} \tag{8.38}$$

Here, it is used that the inner product of the gradient, $\partial f/\partial\boldsymbol{\sigma}$, with respect to the isotropic matrix, $\mathbf{C}$, is a constant, 3μ, see e.g. Krenk (1993a).

Thereby it is possible to formulate the all the relevant methods for a plastic material class using associated von Mises plasticity with linear, isotropic hardening. The behaviour of the von Mises material is dependent on 4 material parameters, no_par=4. The elastic constants, E and ν, must be defined as parameter No. 1 and 2, respectively, in order for the Elastic class to evaluate the constitutive matrix correctly. The remaining two parameters are used to store the yield stress, σ_Y, and the hardening modulus, H.

The loading/unloading condition is determined by the yield function,

yield

```
VonMises.yield(s,a):
   se = equi_stress(s)
   return se – par(3) – par(4)*a
```

The yield function is defined in terms of the equivalent stress, σ_e. A method, equi_stress, uses (8.32) to evaluate the equivalent stress,

equi stress

```
VonMises.equi_stress(s):
   se = 0.5 * ( sqr(s(1)–s(2)) + sqr(s(2)–s(3)) + sqr(s(3)–s(1) )
   se += 3 * ( sqr(s(4)) + sqr(s(5)) + sqr(s(6)) )
   return sqrt(se)
```

The full stress vector is taken as input argument. The gradient of the yield function is found from (8.37).

dyield

```
VonMises.dyield(s,a):
   ss = dev_stress(s)
   se = equi_stress(s)
   for (i=4 to 6) do
      ss(i) = 2*ss(i)
   return 3*ss / (2*se)
```

The method makes use of the deviatoric stress vector. This is found by removing the mean stress from the full stress vector,

dev stress

```
VonMises.dev_stress(s):
   sm = mean_stress(s)
   dev = s
   for (i=1 to 3) do
      dev –= sm
   return dev
```

The mean stress is evaluated from the 3 first terms in stress vector, namely the normal stress terms, $\sigma_{\alpha\alpha}$.

mean stress

```
VonMises.mean_stress(s):
   sm = s(1) + s(2) + s(3)
   return (sm/3)
```

The plastic flow is related to the gradient of the plastic potential. For von Mises associated plasticity the yield function is used as plastic potential. The gradient can therefore be written as

dplast

```
VonMises.dplast(s,a):
   return dyield(s,a)
```

The development of the state parameters, $\boldsymbol{\alpha}$, is related to the derivative of the yield function with respect to $\boldsymbol{\alpha}$. For linear hardening this is a constant, $\partial f/\partial\alpha = -H$, where the hardening modulus, H, is prescribed as material parameter 4, (H = par(4)).

dyield da

```
VonMises.dyield_da(s,a):
   return –par(4)
```

The linear hardening model implies that the hardening function, $h = 1$, simply becomes

hard

```
VonMises.hard(s,a):
   return 1
```

In order to avoid iterations on Gauss point level the explicit radial return will be used to determine the stress increment:

stress inc

```
VonMises.stress_inc(gp)
   c = Elastic.C(gp)
   dse = c * gp.put_dstrain()
   s0 = gp.put_stress()
   a0 = gp.put_state()
   f = yield(s0+dse,a0)
   if (f < 0)
      return dse
   h = par(4)
   mu = par(1) / (2*(1+par(2))
   dl = f / (h + 3*mu)
   ds = dse – 3*dl*mu*dev_stress(s0+dse)/equi_stress(s0+dse)
   da(1) = (equi_stress(s0+ds) - par(3) - a0(1)*h) / h
   gp.set_dstate() = da
   return ds
```

This concludes the implementation of von Mises plasticity with linear isotropic hardening. The methods provide information used by its superclasses Plastic and Material. As each of the methods reflect a small part of the theory they are very simple. It is thus seen that by dividing the properties of the problem into small tasks, an otherwise complex programming task becomes relatively simple with strong possibilities of reuse. The full material hierarchy is given in Figure 8.8.

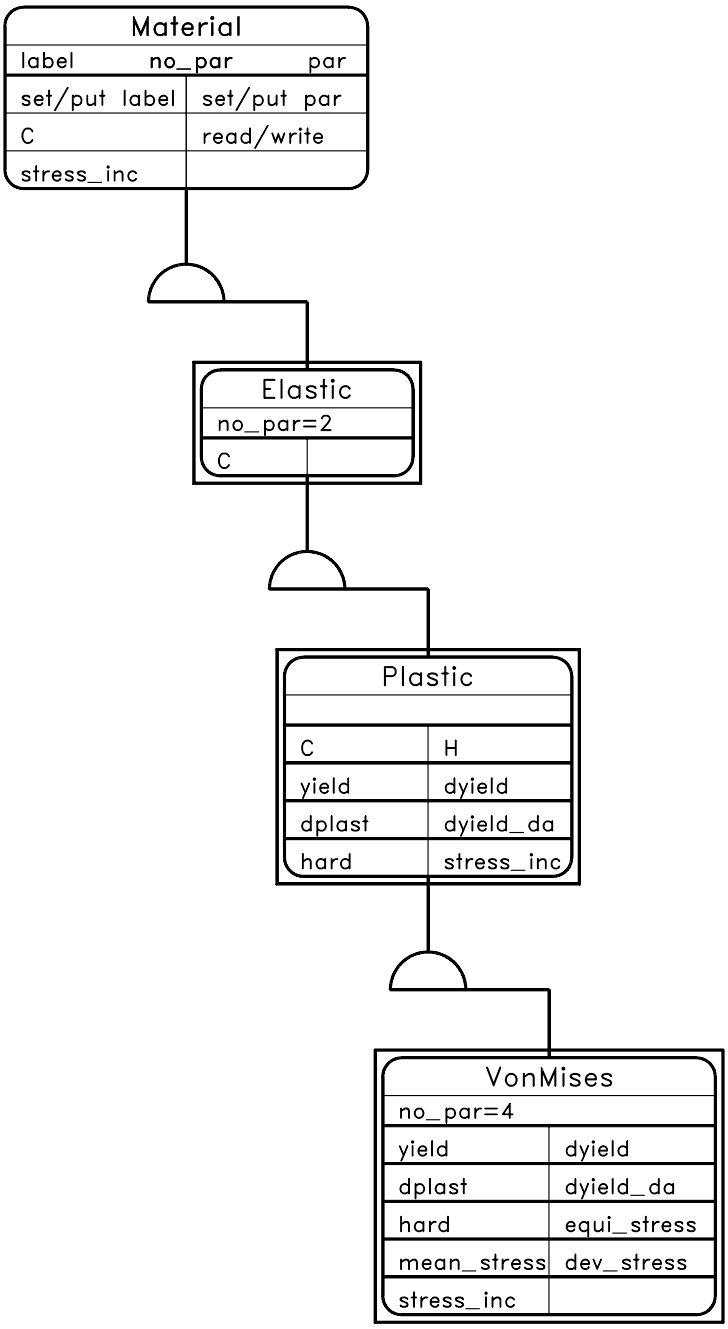


Figure 8.8: Material hierarchy

8.5 Example

The von Mises plasticity model has been used for testing the convergence properties of the orthogonal residual algorithm, Section 6.2, with or without the quasi-Newton correction. The plasticity problem differs from the geometrically non-linear truss problems by being monotonic in the load and thus easier to control. Therefore the example is well-suited for demonstrating the improved convergence properties compared to the modified Newton-Raphson method.

The example uses 8-node 3D solid elements, which are known to experience locking for incompressible elasticity and plasticity, Nagtegaal et al. (1974). Locking is avoided by adding a mean dilatational correction to the strain measure, which imposes the constant volume constraint, see Hughes (1987) pp. 232-237, ABAQUS (1992) *theory manual*.

8.5.1 Example: Plate with hole

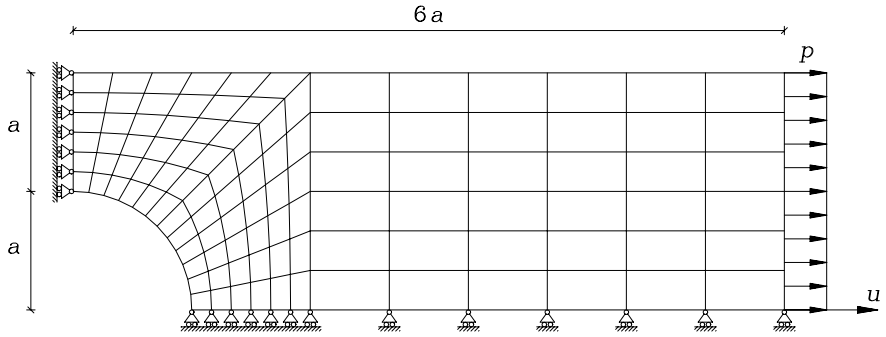


Figure 8.9: Plate with hole - FEM model

A plate with a circular load as in Figure 8.9 is subjected to a uniform axial load with intensity, p. The deformation of the plate is measured by the displacement, u, at the end of the plate. The material properties are Youngs modulus, $E = 2.1\,10^5\,N/mm^2$, Poisson's ratio, $\nu = 0.3$, yield stress, $\sigma_Y = 300\,N/mm^2$, and a hardening modulus, $H = 0$, corresponding to an elastic-perfectly plastic material. Plasticity is modelled as associated von Mises plasticity, cf. Section 8.4. 108 eight-node 3D solid elements are used for discretizing one quarter of the plate. and symmetry boundary conditions are imposed as shown in Figure 8.9.

Convergence is obtained if the combined condition,

$$\|\mathbf{r}\|_{red} < \varepsilon_f \|\Delta\mathbf{f}\|_{red} \quad \wedge \quad \|\delta\mathbf{a}\|_2 < \varepsilon_a a_{max} \tag{8.39}$$

is fulfilled. The convergence thresholds are $\varepsilon_f = 10^{-2}$ on the residual and $\varepsilon_a = 10^{-3}$ on the displacement subincrements. Here, $\|\cdot\|_2$ is the Euclidian norm and $\|\cdot\|_{red}$ is the reduced Euclidian norm, (6.50). The maximum displacement, a_{max}, is defined relative to the linear elastic displacement, thus

$$a_{max} = C\|\mathbf{a}_{elastic}(\Delta\mathbf{p})\|_2 \tag{8.40}$$

with the factor, $C = 4$.

The example uses displacement controlled load incrementation. In each new load step a test increment, $\Delta p = 0.05\,\sigma_Y$, is applied. The first increment, $\Delta\mathbf{a}_1$, is then scaled relative to the converged increment, $\Delta\mathbf{a}_0$, in the previous load step before the equilibrium iterations are performed. In regions with fast or slow convergence it is convenient to change the initial step size, thus a discrete adaptive scheme is applied. If the previous step has converged in less than the desired number of iterations, $i_d = 5$, the first increment is doubled. If instead convergence is not reached within a maximum number of iterations, $i_{max} = 10$, all previous iterations are discarded and the load step is restarted with half the previous increment, see also Section 6.2.2, i.e.

$$\frac{\|\Delta\mathbf{a}_0\|_2}{\|\Delta\mathbf{a}_1\|_2} = \begin{cases} \min\left(2, \dfrac{a_{max}}{\|\Delta\mathbf{a}_1\|_2}\right) & \text{if} \quad i < i_d \\ 1 & \text{if} \quad i_d \le i \le i_{max} \\ \frac{1}{2} & \text{if} \quad i > i_{max} \end{cases} \tag{8.41}$$

This strategy resembles the load incrementation used in the arc-length method and experience has shown that the displacement controlled strategy is more stable than a load controlled strategy used in the examples in Section 7.4.

Load-displacement curves

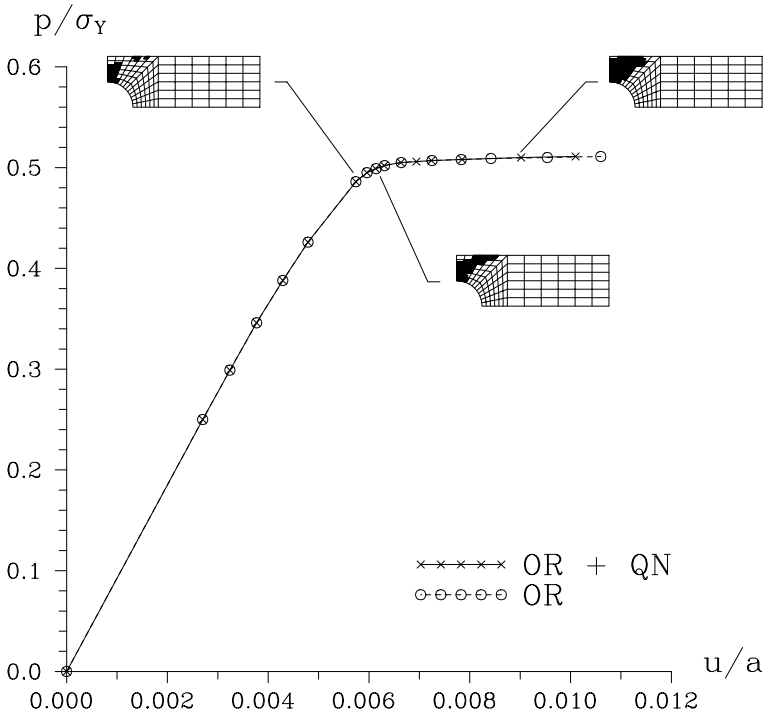


Figure 8.10: Plate with hole - load-displacement curve and development of plastic zones

Figure 8.10 shows load-displacement curves evaluated by the orthogonal residual methods. The first point on the curve marks the elastic limit, and thereafter the plate gradually

becomes plastic until the load capacity is reached. At three selected points the development of the plastic zone is shown.

The first part of the curve represents a slow development of the plastic zone. In this region the elastic part of the stiffness dominates and the linear prediction is representative for the resulting displacements. The iterations are therefore mainly used for correcting the external load, thus the residual part of the convergence condition, (8.39), determines the convergence. The critical points in the analysis are load step 7 and 8. Here, the stiffness changes rapidly because the entire cross section becomes plastic. Therefore it has been necessary to restart with smaller increments. Once these points have been passed the analyses become entirely displacement controlled. Due to the low effective stiffness the changes in the external load becomes relatively small and convergence is then determined by the displacement condition in (8.39). This condition assures an accurate determination of the direction of the displacements. The solution statistics are given in Table 8.2.

Table 8.2: Plate with hole - solution statistics

	OR			OR-QN			MNR
n	$\Delta p/\sigma_Y$	p/σ_Y	i	$\Delta p/\sigma_Y$	p/σ_Y	i	i
1	0.250	0.250	0	0.250	0.250	0	0
2	0.0493	0.299	5	0.0493	0.299	5	5
3	0.0464	0.346	5	0.0464	0.347	5	5
4	0.0425	0.388	5	0.0425	0.388	5	5
5	0.0383	0.426	4	0.0383	0.426	4	4
6	0.0596	0.486	9	0.0596	0.486	9	13
7	0.00906	0.495	20+4	0.00906	0.495	20+4	5
8	0.00391	0.499	10+8	0.00390	0.499	10+9	16
9	0.00257	0.502	3	0.00257	0.502	2	4
10	0.00300	0.505	4	0.00297	0.505	5	9
11	0.00216	0.507	7	0.00131	0.506	5	11
12	0.00128	0.508	7	0.00094	0.507	2	4
13	0.00096	0.509	3	0.00129	0.508	4	8
14	0.00108	0.510	7	0.00153	0.510	7	15
15	0.00068	0.511	7	0.00085	0.511	7	13
		$i_{tot} =$	108			103	117
		$\frac{i_{tot}}{n_{plast}} =$	7.7			7.4	8.4

It is seen that the two versions of the orthogonal residual method (OR: orthogonal residual, OR-QN: orthogonal residaul with quasi-Newton correction) give almost identical results for this problem. The difference occurs in load step 11 where OR uses a doubled first increment because load step 10 has converged in only 4 iterations ($< i_d = 5$). OR-QN uses the original increment size, because the previous load step was completed in 5 iterations. So, the discrete adaptive scheme is sensitive to differences if the number of iterations is close to i_d.

Convergence properties

The convergence properties of the two versions have been examined against the modified Newton-Raphson scheme (MNR). To obtain comparable results the MNR scheme has used the load increments determine by the OR-QN analysis. The average number of iterations is given in Table 8.2 and shows that even though OR and OR-QN uses 3 restarts, i.e. discards 30 iterations, the overall convergence is still better than MNR. Furthermore, the orthogonal residual methods are good both in regions with low and high stiffness, i.e. no matter if it is the load condition or the displacement condition in (8.39) that rules convergence. The convergence rates for three load steps are given in Figure 8.11. It is seen that the orthogonal residual methods, being based on a modified Newton-Raphson scheme, have linear convergence. However, it is found the convergence rates are approximately doubled in regions with low stiffness. The difference between OR and OR-QN is hardly notable. Using the quasi-Newton correction gives a slightly more conservative algorithm, which for some problems would make the solution strategy more robust.

Conclusion

The orthogonal residual method has been used for elasto-plastic analysis of a plate with a circular hole subjected to uniform axial loading. It is found that the algorithms are able to determine the entire load-displacement curve without specfication of predefined load increments. Thus, the discrete adaptive load incrementation scheme works well for the displacement controlled algorithm. The two versions of the orthogonal residual method give almost identical results. Both have linear convergence, which in regions with low stiffness are approximately twice that of the modified Newton-Raphson scheme. This implies that the orthogonal residual method is applicable for elasto-plastic analysis.

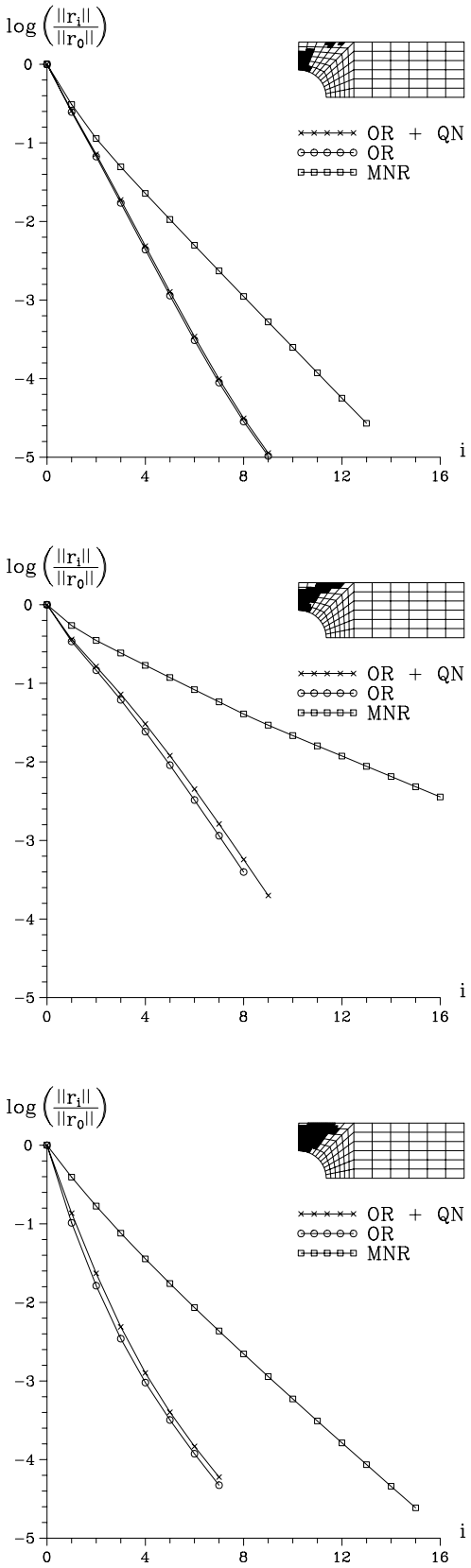


Figure 8.11: Plate with hole - convergence rates

Chapter 9

Conclusion

The aim of this thesis has been to investigate the possibilities offered by object-oriented programming in the development of an open, expandable framework for finite element programming. An object-oriented code, ObjectFEM, has been developed. The program system consists of three levels:

1. Algebraic classes
2. FEM classes
3. Applications

The algebraic classes defines a symbolic syntax for linear algebra and can be used for any scientific programming task. The FEM classes constitute a framework for implementing finite element formulations, which are used by the applications to define an analysis program. In the following the three levels will be considered separately.

9.1 Algebraic classes

The algebraic classes are user-defined data types which can be used like ordinary built-in types. The Vector and Matrix classes are defined for linear algebra developed mainly for programming finite elements. They consist of overloaded arithmetic operators (+,-,*,/) simulating the standard mathematical notation and methods for solution of linear equation systems. The use of the algebraic classes is simple, because the declaration and operator syntax follows the standard for built-in types, hence it is not necessary for the programmer to change the programming style dramatically.

The purpose of introducing the algebraic classes is that the traditional error-prone loops involved in most matrix operations can be replaced by operator calls. This has two consequences: First, the code becomes simpler whereby the programmer is less likely to make errors. Second, the legality of an operation will be tested before it is carried out, thus errors related to mismatching dimensions or out-of-range operations are avoided. For dynamically allocated arrays, as used in C and C++, the class concept offers another important facility: automatic memory control. Traditionally it has been the responsibility of every programmer who uses dynamic memory to ascertain that allocated memory is freed correctly when it is no longer in use. For an object of a class, however, the allocated

memory is automatically freed, because the program itself calls a method that reclaims the memory - the destructor.

The numerical efficiency of the classes is not reduced compared to standard C, because the internal operations manipulate the arrays directly. Still, for operations where one of the arguments can not be overwritten it is necessary to create a temporary object which finally is copied to the global scope. These operations imply that memory is allocated and freed more often as would have been the case if the operation was carried out explicitly in the program. For operations on small matrices this does generally not give any computational overhead, but for larger matrices it should be considered whether a traditional looping technique should be employed.

The algebraic classes are defined in such a way that the programmer is able to modify them to the current needs, e.g. by improving an existing operator or method, or by adding a new one. This requires the programmer to modify the implementation of the class. Using the existing code as model for the modifications the task is reasonably simple for programmers with a basic knowledge of C++. Furthermore, the new version of the class can be fully tested before it is linked to the existing part of the system. The changes do not affect the existing codes that employ the classes, hence these can be used without being modified.

9.2 FEM classes

The FEM classes constitute a framework for programming the finite element formulations. Its kernel is the base classes notably Node, Element and Material. The base classes define an interface consisting of shared methods and FEM methods. The shared methods take care of tasks that are common to all problems, e.g. model definition, generation and input/output. To make these methods work a number of problem parameters is introduced mainly for bounding the internal loops and sizing the internal arrays. This part of the class interface thus consists of parametric methods - a technique used in procedural programming as well.

The FEM methods specify the finite element formulation, e.g. the element stiffness or strain, and they are to be implemented for each new element or material type. A new element is introduced as a subclass of the base class, Element. It inherits the shared methods from the base class, thus by setting the problem parameters it can use an already implemented shared methods. The subclass also inherits the declaration of the FEM methods from the base class. For an element these are methods for evaluating the stiffness, load, strain and stress. The programmer must interpret the current finite element formulation in terms of these four methods. For a bar element the stiffness, strain and stress can be evaluated explicitly, hence its implementation involves only a few additional methods, e.g. for evaluating the directional vector of the bar. Isoparametric continuum elements use shape functions and numerical integration in the compuation of the element properties. It is therefore necessary to define a larger number of methods and to introduce a Gausspoint class to manage the coordinates and weights of each integration point. The number and type of additional methods vary from one problem to another. The base class, Element, is therefore a programming framework where it is the responsibility of the programmer to sup-

ply the entire finite element formulation, rather than a standard scheme for programming finite elements. However, this combination of parametric shared methods and inherited FEM methods enables the programmer to concentrate on the problem formulation.

For some element types and material models it is possible to obtain full advantage of the inheritance concept. An isoparametric continuum element is mainly described by the order of its shape functions. The specification of a new element thus consists of defining a shape function matrix, a gradient matrix and the integration order, while the remaining parts, e.g. computation of the stiffness matrix, can be inherited from an isoparametric superclass.

In the present work bar elements and isoparametric continuum elements have been implemented. It is demonstrated how the different formulations can be captured by an interface consisting of only four methods. Comparing this to the systems presented by others, see e.g. Section 1.4, it is found that the common features reduce to these four concepts, thus instead of providing a standard scheme for programming finite elements, object-oriented programming is used basically to structure the code into modules that are highly independent of each other. This makes it possible to add new elements or material models without changing the existing code. Furthermore, elements and materials inherited from the base classes reuse existing code, hence reducing the size of the total code.

The programmer introduces new elements or materials by implementing a subclass. If, for example, the new element belongs to a group of existing elements, e.g. isoparametric elements, the implementation will usually consist of redefining a few methods. This can be done with an existing class as model and is therefore mainly a matter of getting the formulation correct. If instead a new problem type is to be introduced, e.g. shells or plates, the programmer will have to provide the entire formulation, including both element and material. It requires the programmer to consider which additional attributes and methods are needed. If it is not likely that other elements of this type are later implemented the programmer can simply implement the additional methods as in a traditional program. If, however, the element is part of a group of elements of which others are likely to be introduced later on, the programmer should consider defining the methods so that they can be inherited by subclasses. This will require the programmer to be familiar with the inheritance concept and its implementation in C++. Considering ObjectFEM mainly as a laboratory for finite element programming the author finds that the first technique where inheritance is not taken into account should be employed. In this way the programmer may become familiar with the system without having to know much about object-oriented programming.

9.3 Applications

The algebraic classes and FEM classes provides a macro-language for programming a finite element analysis program - an application. An application consists of model definition and generation, formation and solution of the global equation system, and postprocessing. The model definition and generation take care of model input. In this phase the available element types or material models must be known so it is possible to typecast the object in

order to activate the appropriate version of the methods. In the following no distinction is made between different element or material types. The formation and solution of the global equations form the solution strategy. In the present work there are applications for linear and non-linear analyses. They are written in terms of Elements and Nodes, i.e. without reference to the specific formulation. This enables the user to analyze different problem types with the same application. Also the use of the macro-language gives a compact code consisting of standard loops for communication with Element and Node objects and otherwise linear algebra handled by the algebraic classes. It is the experience that this macro-language makes it easy to program new solution algorithms. The application uses existing facilities, thus the programmer is only required to learn some basic constructs for manipulation of the finite element model, but not the object-oriented programming style. Having an existing program as model it is therefore simple to make a new application - especially if it is possible to reuse the input module that defines the model.

9.4 An open, expandable framework

Object-oriented programming has been used for structuring finite elements. It is found that the programmer benefits in two ways. First, the finite element analysis can be divided into three levels: linear algebra, finite element formulation and solution strategies. In object-oriented programming this structure gives a distributed architecture where the different parts can be developed independently of each other. This enables the programmer to modify one part of the program without affecting the other parts. Second, the finite element formulation relates to three concepts: node, element and material. These concepts can be used to define base classes in an object-oriented structure. The base classes do not define a standard scheme for implementing elements and materials, but provide a framework where only the finite element formulation is handled by the programmer. Objects are thus used for structuring the code and through inheritance they enable reuse of a major part of the code.

The aim has been to define an open, expandable framework for easy prototyping of new elements, material models and solution methods. Three levels of programming have been introduced: algebraic classes, FEM classes and applications. The algebraic classes can immediately be used by programmers with experience in traditional scientific programming to produce clear and readable code. The FEM classes constitute a framework for fast prototyping of elements and materials, but requires the programmer to be familiar with the base class definition and have a general understanding of object-oriented programming (in C++). The algebraic classes and FEM classes provide a macro-language for programming finite element analysis applications. The applications use the existing facilities, hence requires the programmer to learn a few basic constructs.

The major inhibiting factor in the development and presentation of an object-oriented framework for finite element programming has been that the object-oriented principles are still unfamiliar to most programmers of scientific codes. Changing to a fully object-oriented framework not only requires the programmer to learn a new programming syntax but also a new programming style. This is of course not optimal. A way to obtain some

of the benefits of the object-oriented programming style would be to introduce the tools, e.g. algebraic classes, into the procedural programs and then gradually to use more of the object-oriented facilities.

Chapter 10

References

ABAQUS (1992): *ABAQUS Vers. 5.2 – manuals.* Hibbitt, Karlsson & Sorensen Inc.

ANSYS (1988): *ANSYS-PC/Linear 4.3 – reference manual.* Swanson Analysis Systems Inc.

Baugh Jr., J.W. and Rehak, D.R. (1989): Object-Oriented Design of Finite Element Programs. In *Computer Utilization in Structural Engineering.* (ed. Nelson, J.K.), pp. 91-100, San Francisco, USA, 1989.

Baugh Jr., J.W. and Rehak, D.R. (1992): Data Abstraction in Engineering Software Development. *Journal of Computing in Civil Engineering.* Vol. 6, No. 3, pp. 282-301, 1992.

Beltzer, I.A. (1990): *Variational and Finite Element Methods: SMC-Approach.* Springer Verlag, 1990.

Bergan, P. (1980): Solution algorithms for nonlinear structural problems. *Computers and Structures.* Vol. 12, pp. 497-509, 1980.

Bergan, P. (1981): Solution by iteration in displacement and load spaces. In *Nonlinear Finite Element Analysis in Structural Mechanics.* (eds. Wunderlich, W., Stein, E. and Bathe, K.-J.), pp. 553-571, Springer Verlag, Berlin, Germany, 1981.

CALFEM (1993): *CALFEM: Computer Aided Learning of the Finite Element Method Vers. 3 – manual.* Technical University of Lund, Lund, Sweden, 1993.

Chen, W.F. and Han, D.J. (1988): *Plasticity for Structural Engineers.* Springer Verlag, New York, USA, 1988.

Coad, P. and Yourdon, E. (1991): *Object-oriented design.* Prentice-Hall, New Jersey, USA, 1991.

Crisfield, M.A. (1981): A fast incremental/iterative solution procedure that handles 'Snap-through'. *Computers & Structures.* Vol. 13, pp. 55-62, 1981.

Crisfield, M.A. (1991): *Non-linear Finite Element Analysis of Solids and Structures – Volume 1: Essentials.* John Wiley & Sons, Chichester, U.K., 1991.

Dahlblom, O., Peterson, A. and Petersson, H. (1985): *CALFEM – Ett datorprogram för undervisning i finita elementmetoden.* Technical University of Lund, Lund, Sweden, 1985. (in swedish)

Dubois-Pèlerin, Y. (1992): Object-Oriented Finite Element Programming: Programming concepts and implementation. *Ph.D. thesis at Ecole Polytechnique Federale de Lausanne*, Thesis No. 1026, 1992, Lausanne, Switzerland, 1992.

Dubois-Pèlerin, Y., Zimmermann, T. and Bomme, P. (1992): Object-Oriented Finite Element Programming. II: A Prototype Program in Smalltalk. *Computer Methods in Applied Mechanics and Engineering.* Vol. 98, pp. 361-397, 1992.

Dubois-Pèlerin Y. and Zimmermann, T. (1993): Object-Oriented Finite Element Programming. III: An efficient implementation in C++. *Computer Methods in Applied Mechanics and Engineering.* Vol. 108, pp. 165-183, 1993.

Ellis, M.A., and Stroustrup, B. (1990): *The Annotated C++ Reference Manual.* Addison-Wesley, Massachusetts, USA, 1990.

Forde, B.W.R., Foschi, R.O. and Stiemer, S.F. (1990): Object-Oriented Finite Element Analysis. *Computers & Structures.* Vol. 34, No. 3, pp. 355-374, 1990.

Hededal, O. (1993): Finite element with C++ classes. *Proc. 6th Nordic Seminar on Computational Mechanics.* Linköping, Sweden, 1993.

Hededal, O. and Krenk, S. (1993): A Profile Solver in C for Finite Element Equations. *Engineering Mechanics Papers.* No. 13, Dept. Building Technology and Structural Engineering, Aalborg University, Aalborg, Denmark. (to appear in *Computers & Structures*)

Hughes, T.J.R. (1987): *The Finite Element Method – Linear static and dynamic analysis.* Prentice-Hall, New Jersey, USA, 1987.

Kernighan, B. and Ritchie, D.M. (1991): *The C Programming Language, 2nd Edn.* Prentice Hall, New Jersey, USA, 1991.

Krenk, S. (1993a): *Non-linear analysis with finite elements.* Dept. Building Technology and Structural Engineering, Aalborg University, Aalborg, Denmark, 1993.

Krenk, S. (1993b): An Orthogonal Residual Procedure for Nonlinear Finite Element Equations. *Engineering Mechanics Papers.* No. 18, Dept. Building Technology and Structural Engineering, Aalborg University, Aalborg, Denmark, 1993. (to appear in *International Journal of Numerical Methods in Engineering*)

Krenk, S. and Hededal, O. (1993): A Dual Orthogonality Procedure for Nonlinear Finite Element Equations. *Engineering Mechanics Papers.* No. 21, Dept. Building Technology and Structural Engineering, Aalborg University, Aalborg, Denmark, 1993. (submitted for publication)

Krieg, R.D. and Krieg, D.G. (1977): Accuracies of numerical solution methods for the elastic-perfectly plastic model. *ASME Journal of Pressure Vessel Technology.*, Vol. 99, pp. 510-515, 1977.

Lippman, S.B. (1989): *C++ Primer.* Addison-Wesley, Massachusetts, USA, 1990.

Luenberger, D.G. (1984): *Linear and Nonlinear Programming - 2nd Edn.* Addison-Wesley, Massachusetts, USA.

Mackie, R.I. (1992): Object oriented programming of the finite element method. *International Journal for Numerical Methods in Civil Engineering.* Vol. 35, pp. 425-436, 1992.

Matthies, H. and Strang, G. (1979): The solution of nonlinear finite element equations. *International Journal for Numerical Methods in Engineering.* Vol. 14. pp. 1613-1626, 1979.

Menétrey, P. and Zimmermann, T. (1992): Object-oriented non-linear finite element analysis: application to J2 plasticity. Internal report IBAP: 92.03:01, LSC: 92.25. Dept. of Civil Engineering, Swiss Federal Institute at Lausanne, Lausanne, Switzerland, 1992.

Miller, G.R. (1991): An object-oriented approach to structural analysis and design. *Computers & Structures.* Vol. 40, No. 1, pp. 75-82, 1991.

Nagtegaal, J.C., Parks, D.M. & Rice, J.R. (1974): On Numerically Accurate Finite Element Solutions in the Fully Plastic Range. *Computer Methods in Applied Mechanics and Engineering.* Vol. 4, pp. 153-177, 1974.

Nielsen, L.O. (1993): A C++ basis for computational mechanics software. Technical University of Denmark, Copenhagen, Denmark, 1993.

Ottosen, N.S. (1987): Aspects of constitutive modelling. Report TVSM-3010, Lund Institute of Technology, Lund, Sweden, 1987.

Ottosen, N.S. and Petersson, H. (1992): *Introduction to the Finite Element Method.* Prentice Hall, U.K., 1992.

Press, W.H., Flannery, B.P., Teukolsky, S.A. and Vetterling, W.T. (1988): *Numerical Recipes in C.* Cambridge University Press, Cambridge, U.K., 1988.

Ramm, E. (1981): Strategies for tracing the non-linear response near limit-points. In *Nonlinear Finite Element Analysis in Structural Mechanics.* (eds. Wunderlich, W., Stein, E. and Bathe, K.-J.) pp. 63-89, Springer Verlag, Berlin, Germany, 1981.

Riks, E. (1979): An incremental approach to the solution of snapping and buckling problems. *International Journal of Solids and Structures.* Vol. 15, pp. 529-551, 1979.

Ristinmaa, M. (1993): Cyclic Plasticity and its Numerical Treatment. *Ph.D. thesis at Lund Institute of Technology.* Department of Solid Mechanics, Lund Institute of Technology, Lund, Sweden, 1993.

Ross, T.J., Wagner, L.R. and Luger, G.F. (1992a): Object-Oriented Programming for Scientific Codes. I: Thoughts and Concepts. *Journal of Computing in Civil Engineering.* Vol. 6, No. 4, pp. 480-496, October, 1992.

Ross, T.J., Wagner, L.R. and Luger, G.F. (1992b): Object-Oriented Programming for Scientific Codes. II: Examples in C++. *Journal of Computing in Civil Engineering.* Vol. 6, No. 4, pp. 497-514, 1992.

Schwarz, H.R. (1988): *Finite Element Methods.* Computational Mathematics and Applications series, Academic Press, London, U.K., 1988.

Scholz, S.-P. (1992): Elements of an Object-Oriented FEM++ Program in C++. *Computers & Structures.* Vol. 43, No. 3, pp. 517-529, 1992.

Stroustrup, B. (1991): *The C++ Programming Language, 2nd Edn.* Addison-Wesley, Massachusetts, USA, 1991.

Winder, R. (1991): *Developing C++ Software.* John Wiley & Sons Ltd, Chichester, U.K., 1991.

Yang, Y.-B and Leu, L.-J (1991): Constitutive laws and force recovery procedures in nonlinear analysis of trusses. *Computer Methods in Applied Mechanics and Engineering.* Vol. 92, pp. 121-131, 1991.

Yu, G. and Adeli, H. (1993): Object-Oriented Finite Element Analysis using EER Model. *Journal of Structural Engineering* Vol. 119, No. 9, pp. 2763-2781, 1993.

Zienkiewicz, O.C. and Taylor, R.L. (1989): *The Finite Element Method Vol. 1 – Basic Formulation and Linear Problems.* McGraw-Hill, London, U.K., 1991.

Zienkiewicz, O.C. and Taylor, R.L. (1991): *The Finite Element Method Vol. 2 – Solid and Fluid Mechanics, Dynamics and Non-Linearity.* McGraw-Hill, London, U.K., 1991.

Zimmermann, T., Dubois-Pèlerin, Y. and Bomme, P. (1992): Object-Oriented Finite Element Programming. I: Governing Principles. *Computer Methods in Applied Mechanics and Engineering.* Vol. 98, pp. 291-303, 1992.

Appendix A

Algebraic classes

An important part of scientific programming is linear algebra. The mathematical notation such as used in this thesis, provides an compact notation for operations that would otherwise be difficult to overview. In C++ the algebraic operators can be simulated to give a symbolic programming style that lies close to the mathematical notation.
This chapter presents a set of algebraic classes. The floating point classes, Matrix, ProMatrix and Vector, are used for the matrix manipulations and integer classes, IntArray and Int2DArray, are array structures used to store information e.g. boolean variables. The classes are mainly developed for writing finite element codes, therefore they presently only contain the algebraic operators, `+`, `-`, `*` and `/`, and methods for vector calculus and solution of linear equation systems.
C++ provides the object-oriented facilities, i.e. operator overloading and polymorphism, which enables binding of functionally to a data structure, see e.g. Stroustrup (1991), Lippman (1989), Winder (1991). C++ is therefore used for programming the interface which handles the communication between an object and the other parts of a program. The numerical part of the classes are programmed in standard C, whereby the numerical efficiency of C is retained. Principles for scientific programming in C are given e.g. by Press et al. (1988)
In Section A.1 the different parts of the class definition are considered, describing some of the principles used in the development of the classes. The implementation of and programming with overloaded operators is given in Section A.2. Algorithms for solving the linear equation system, which is a central part of finite element codes, are considered in Section A.3. The full class declarations are then given in Table A.1-A.5. Section A.4 gives five examples that illustrate the use of the classes in scientific programming. A short reference list concludes the chapter.
The C++ program code is given in `Courier` typeface. The code segments are divided in *Implementation* and *Syntax* giving examples of their use in the program.

A.1 Class declaration

The class declaration consists of attributes, constructors/destructor, methods and operators. The attributes are encapsulated by declaring them `private`, whereby they can only be manipulated using the `public` part of the class.

class declaration

```
class Vector
{
  private:
    Attributes:
    int no_row;
    double *v;

  public:
    Constructors & destructor:
    Vector();
    Vector(const Vector& b);
    Vector(int nr);
    ~Vector();
    Operators:
    Vector& operator = (const Vector& b);
    Vector operator + (const Vector& b);
        ⋮
    Member methods:
    int size(int i);
        ⋮
    Friend methods:
    friend double length(const Vector& b);
        ⋮
}
```

A.1.1 Attributes

An object of an algebraic class is defined by the number of rows (`no_row`) and the number of columns (`no_col`). The body of the object is stored by a pointer variable, `*v` or `**m`, for which the memory is allocated and freed dynamically. A standard C pointer technique for dynamic memory allocation is used to create the one- or two-dimensional array. The standard matrix index scheme is retained, such that the rows are indexed from `1` to `no_row`, and the columns are indexed `1` to `no_col`, see e.g. Press et al. (1988). The ranges of the arrays are stored by the two attributes, `no_col` and `no_row`. This enables verification of the validity of an operation. During assignment the object can perform a range check, thus preventing reading or writing in out-of-range memory. The algebraic operations require the dimensions of the two objects to match and a comparison of the dimensions may be used to detect illegal operations, e.g. if the dimensions of two matrices match when a multiplication is performed.

A.1.2 Constructor and destructor

User-defined types, such as the algebraic classes, are declared and used like ordinary built-in types, e.g. characters, integers or floating point variables. However, in order to behave like the built-in types the user-defined types must have constructors and a destructor.
Before a variable is used it must be declared by type and name and possibly be initialized. The type-casting tells the compiler to allocate a specified amount of memory, which is used for storing the variable. During the declaration the variable is also initialized either by a default value or a value specified by the user. The declaration operation uses a constructor. The constructor is responsible for allocating memory and initializing the variable. For user-defined types these operations are not predefined, so the programmer must provide their implementation. For the algebraic classes construction consists of setting the size of the array and allocating memory. Because the variables can be initialized in different ways, it is generally necessary to provide a variety of constructors. There are two essential ones which should always be defined: the default constructor and the copy constructor. The default constructor is used to declare uninitialized objects,

default constructor

Implementation:

```
Vector::Vector()
{
  no_row = 0;
  v = 0;
}
```

Syntax:

```
Vector a;
```

The binding to the class is stated explicitly in the implementation, e.g. `Vector::Vector()`, where the operator `::` specifies that the constructor is part of the prefixed class.
The copy constructor is called whenever the variable is copied, e.g. between the main program and a subroutine. It takes another object as argument and creates a copy of it.

copy constructor

Implementation:

```
Vector::Vector(Vector& b)
{
  v = new double [no_row = b.no_row];
  v--;
  for (int i=1; i<=no_row; i++)
    v[i] = b.v[i];
}
```

Syntax:

```
Vector b(a);
```

Usually the constructor is called automatically by the program in order to copy an object from one part of the program to another. Still, it is possible to call the constructor

explicitly to initialize a new object. The statement, `b(a)`, handles both declaration and initialization. First the vector, `b`, is declared and then initialized by the argument, `a`, which is a predefined vector for which the size is known and the memory is already allocated. The constructor first allocates `no_row` fields of double precision floating points by use of the `new` operator. In order to maintain indexing from 1 to `no_row`, the pointer is shifted back by 1, `v--`, before it is initialized. Finally, it copies each member of the argument to the new object. Usually the size of the object will be known upon declaration, thus a third constructor is introduced which sets the size, allocates memory and initializes all members to 0,

——— *constructor* ———

Implementation:

```
Vector::Vector(int nr)
{
  v = new double [no_row = nr];
  v--;
  for (int i=1; i<=no_row; i++)
    v[i] = 0;
}
```

Syntax:

```
Vector a(6);
Vector b;
b = Vector(3);
```

As for the copy constructor, this constructor handles both declaration and initialization. It first declares the vector and then initializes the object according to the argument. The program is able to distinguish between the two types of constructors by looking at the argument type, thus the constructors are polymorph, see e.g. Stroustrup (1991). This constructor is also used for allocating memory to objects, e.g. `b`, declared by the default constructor. In that case the program makes use of the assignment operator, `=`, described in the following section.

A variable is only active within the scope where it is defined. The scope can be the entire program, a subroutine or a loop. When a variable goes out of scope, the allocated memory should be freed, so that other parts of the program can reuse it. This operation, which is predefined for the built-in types, is handled by the destructor. The user-defined type must provide a destructor that safely frees the allocated memory.

——— *destructor* ———

Implementation:

```
Vector::~Vector()
{
  if (++v) delete (v);
}
```

Before freeing the memory held by the pointer, it is moved to its original position, i.e. `++v`, and it is tested whether any memory has been allocated. The destructor is called

automatically when the variable goes out of scope and is therefore seldom called explicitly. The use dynamically allocated memory is traditionally a sensitive part of a C program, mainly because there is not any facility for performing range check. The constructor-destructor technology enables safe manipulation with dynamic variables such as pointers, because it is in the hands of the object itself to allocate and free the memory.

A.1.3 Member methods and friend methods

In object-oriented programming the attributes and methods are gathered in an entity - the class. The methods are used for manipulation of the attributes. The attributes are usually encapsulated, i.e. hidden from the other parts of the program, and can only be manipulated by the class methods. In C++ the methods that manipulate the attributes are divided in two: member methods and friend methods. The member methods are part of the each object, which keep their reference in a table along with the reference to the attributes. The binding to the class is stated explicitly in the implementation, e.g. `Vector::length()`, where the operator `::` specifies that the method is part of the prefixed class.

member method

Implementation:

```
double Vector::length()
{
  double l=0;
  for (i=1; i<=no_row; i++)     l += sqr(v[i]);
  return sqrt(l);
}
```

Syntax:

```
Vector b(4);
double l = b.length();
```

The use of a member method usually corresponds to that of sending a message in pure object-oriented languages, thus should ideally not take any arguments. This is also reflected in the calling syntax, where the object is prefixed to the method name, e.g. `b.length()`. The friend methods are external methods that are allowed to access the encapsulated attributes directly. They are, however, not part of any object and must therefore include the objects in the argument list. Alternatively, the length of the vector is found using a friend method:

friend method

Implementation:

```
double length(const Vector& b)
{
  double l=0;
  for (i=1; i<=b.no_row; i++)
    l += sqr(b.v[i]);
  return sqrt(l);
}
```

Syntax:

```
Vector b(4);
double l = length(b);
```

The choice whether to use a member method or a friend method is mainly a matter of programming style. In the present framework most of the methods are defined as friends. Exceptions are the `size` method that gives the dimensions of the arrays and the `T` method which returns the transpose of a rectangular matrix, i.e.

transpose

Syntax:

```
Matrix A(4,5), B;
B = A.T();
```

A method can be a friend of several classes, e.g. of the Vector and the Matrix class. This is e.g. the case for the solution method, solve, which take both a Matrix and a Vector as arguments. In order to make the operations efficient it is declared as a friend to both the Matrix and Vector class, allowing the method to access the private attributes of both objects directly. Even though the friend methods are external methods independent of the individual objects of a class, they are usually closely related to the implementation of the classes which they manipulate. Therefore they should be defined in conjunction with the classes.

A.1.4 Arguments and return values

In C++ data are usually passed from one part of a program to another in two ways: by copy or by reference. By copying the variable every time it is passed to another part of the program it is ensured that the global variable is not altered. The copy may either be passed to a method through the argument list or passed from the method as a return value.

pass a copy

Implementation:

```
double det(Matrix A)
{
  double determinant = 1;
  if (no_row==no_col)
  {
    factor(A);
    for (i=1; i<=no_row; i++)
      determinant *= A.m[i][i];
  }
  return determinant;
}
```

Syntax:

```
Matrix A(5,5);
double determinant = det(A);
```

The matrix `A` is copied into the method, where it is factorized in order to calculate the determinant. The result, `determinant`, copied to the global scope. The global matrix remains unaffected even though the local matrix has been factorized.
For small objects the computational overhead involved in passing a copy of the entire object is usually acceptable. Copying larger objects such as matrices the excessive number of operations leads to unacceptable computational overhead and instead these should be passed as reference. Passing a reference means that the program creates an address variable that points at the data field of the global variable. This address is copied to the method that can access the data fields.

pass a reference

Implementation:

```
double length(Vector& b)
{
  double l=0;
  for (i=1; i<=no_row; i++)
    l += sqr(b.v[i]);
  return sqrt(l);
}
```

Syntax:

```
Vector b(4);
double l = length(b);
```

The vector, `b`, is passed to the method by reference, which is indicated by the `&`-suffix on the argument type, e.g. `Vector&`. It is seen that neither in the call nor in the method it is possible to distinguish between an argument passed by copy and passed by reference. As it is the address that is copied the method is able to alter the global value of the variable. To ensure that the argument remains unaltered by the method it can be defined as a constant, i.e. `double length(const Vector& b)`.
The reference technique can also be used when returning values from a method. Like

for the arguments it can be convenient to return references instead of copies, because it is more efficient. The use of this technique is limited to those return values that either involve global variables, or attributes in the object. If a temporary, local variable is not copied out, it will be destroyed upon return from the method, thus the value stored by the reference address can be overwritten by the other parts of the program.

A.1.5 Coercion

The Matrix class can be specialized by precision e.g. integer or double precision floating point. It can also be specialized according to its structure, distinguishing between full non-symmetric, rectangular matrices, triangular matrices or profile matrices (skyline). The symmetric, square matrices could be derived from a Matrix superclass using virtual methods for nearly all operators and methods. This is, however, less efficient than static binding and therefore a different style has been chosen, namely a type coercion technique. Coercion is an implicit operation that e.g. enables the program to add a integer variable to a floating point variable: the integer variable is first converted to a floating point so the floating point operation can be performed. Coercion for the base types is a built-in facility in most languages.
In C++ it is possible to define coercion for the user-defined types as well. This implies that only those operations that do not alter the matrix structure should be defined for the specialized matrix classes. E.g. adding 2 symmetric matrices produces a new symmetric matrix, thus this operation should be defined explicitly for symmetric matrices. Multiplication of 2 symmetric matrices gives in general a full non-symmetric matrix and in this case the operation can be performed by the general full matrix class which by use of a coercion rule converts the symmetric matrix to a full matrix. The coercion operator defines the conversion from a ProMatrix to a full rectangular Matrix, i.e.

coercion

Implementation:

```
ProMatrix::operator Matrix()
{
  Matrix A(n_row,n_row);
    for (i=1; i<=n_row; i++)
      for (j = p(i)+1; j<=i; j++)
        A(i,j) = A(j,i) = m[i][j];
  return A;
}
```

Syntax:

```
ProMatrix A(4);
Matrix B, C;
B = (Matrix)A;
C = A;
```

The operator can be evoked explicitly as for the `B`, but in most cases the program implicitly evokes the operator, e.g. in the assignment `C = A`. The full matrices are copies of the original matrix, `A`, which therefore remains a profile matrix.

A.2 Operators

The introduction of the algebraic data types, Matrix and Vector, is based on the possibility to retain the mathematical notation style in the programming of scientific codes by use of overloaded operators. The overloading of an operator consists of providing an implementation, that simulates the behaviour of the operator with which it is associated in the current context. E.g. the addition of two matrices, $\mathbf{C} = \mathbf{A} + \mathbf{B}$, should have a direct counterpart in the program, i.e.

operator

Syntax:

```
Matrix A(3,4), B(3,4), C;
C = A + B;
```

This operation requires two operators to be defined: assignment, `=`, and addition, `+`. An operator is declared and implemented like an ordinary method - either as member or friend. The calling syntax is, however, the same as for the predefined operators.
The algebraic classes have operators for assignment, arithmetic operations, input and output. In the following examples of the implementations and syntax is presented. The declarations of all the operators are given in Table A.1-A.5.

A.2.1 Assignment

There are two types of assignment operation related to algebraic classes: assignment of another object and assignment of a single member. The assignment operator, `=`, resembles the copy constructor. The constructor creates a new object as a copy of the argument. The assignment operator, however, replaces the original object by a copy of the argument, i.e.

assignment

Implementation:

```
Vector& Vector::operator = (Vector& b)
{
  if (v == b.v)
    return *this;
  if (++v) delete (v);
  v = new double [no_row = b.no_row];
  v--;
  for (int i=1; i<=no_row; i++)
    v[i] = b.v[i];
  return *this;
}
```

Syntax:

```
Vector a(3), b;
b = a;
```

The operator compares the addresses of the arrays, `v` and `b.v`. If these are the same, i.e.

if it is attempted to assign an object to itself, the operator just returns the self-reference, `*this`. Otherwise, the old array is deleted and a new created as a copy of the argument. It is in many cases necessary to access the single element in the array. This can be done through the index operator, `()`, which takes the index as argument and returns a reference to the member. Using a reference as return value enables the index operator to be used both for assignment and access of the single member.

——— *index operator* ———

Implementation:

```
double& Vector::operator(int i)
{
  if (v && i>=1 && i<=no_row)
    return v[i];
}
```

Syntax:

```
double x;
Vector b(3);
b(1) = 1;
x = b(3);
b(4) = 5.5;      // Illegal operation
```

The last operation is out-of-range and is therefore not carried out. The index operator thus prevents reading and writing in out-of-range memory. It should be noted that accessing an element in the array through the index operator is time consuming compared to direct access of the pointer, see e.g. Nielsen (1993). Procedures or methods that involve many single-element assignments should therefore be able to operate directly on the array pointer. This could be done by declaring the method as a friend. Still, this would require the class declaration to be modified for each new method and instead a method, `pointer`, is defined which returns the array pointer.

——— *access pointer* ———

Implementation:

```
double* Vector::pointer()
{
  return v;
}
```

Syntax:

```
Vector b(3);
double *b_ptr = b.pointer();
for (i=1; i<=3; i++)
  b_ptr[i] = i;
```

The global pointer variable, `b_ptr`, is initialized with the pointer to the vector array, `b.v`. The two variables thus point at the same location in the memory. The assignment operation, `b_ptr[i]=i`, therefore is equivalent to `v(i)=i`. The vector thus becomes $\mathbf{b} = [1 \;\; 2 \;\; 3]$. The disadvantage of this technique is of course that there is no automatic range

check.

A.2.2 Arithmetic operators

The arithmetic operators for the algebraic classes are `+`, `-` and `*`. Each of these has a corresponding operator with assignment, i.e. `+=`, `-=` and `*=`, where the result is assigned to the calling object, `this`. For example, the addition with assign of two vectors is defined in the following way:

addition w. assign

Implementation:

```
Vector& Vector::operator += (Vector& b)
{
  if (no_row == b.no_row)
    for (i=1; i<=no_row; i++)
      v[i] += b.v[i];
  return *this
}
```

Syntax:

```
Vector u(4), v(4);
u += v;
```

The operator first checks the validity of the operation. The operation then consists of adding each member in the argument vector, `b.v[i]`, to the body of the calling object, `v[i]`. This enables a simple definition of the addition operator,

addition

Implementation:

```
Vector Vector::operator + (Vector& b)
{
  Vector a(*this);
  a += b;
  return a;
}
```

Syntax:

```
Vector u, v(4), w(4);
u = v + w;
```

The statement first performs the addition, `v + w`, and then assigns the result to a new vector, `u`. The addition operator is defined as a member method with one argument. This means that the operation, `v + w`, is interpreted as `v.operator+(w)`. The operator creates a copy of the calling object using the copy constructor. Then the operator, `+=`, is called adding the argument to the copy.
Multiplication of two matrices - symmetric or un-symmetric - generate a full un-symmetric matrix. It is therefore chosen only to implement the multiplication operator for the Matrix class. The multiplication operator, `*`, is defined as a friend method, which takes the two

matrices as argument. If the multiplication involves a profile matrix the coercion operator, `ProMatrix::Matrix()`, is called, converting the argument to a full matrix. Thereby it is not necessary to define the multiplication operator in the ProMatrix class. For sparse matrices this technique is of course inefficient because it uses elements that are 0 and the operator should be defined for this type of matrix. This is not done in the present implementation.

——— *multiplication* ———

Implementation:

```
Matrix operator * (const Matrix& A, const Matrix& B)
{
  if (A.no_col!=B.no_row) return Matrix();

  Matrix C(A.no_row,B.no_col);
  for (i=1; i<=A.no_row; i++)
    for (j=1; j<=B.no_col; j++)
      for (k=1; k<=A.no_col; k++)
        C.m[i][j] += A.m[i][k] * B.m[k][j];
  return C;
}
```

Syntax:

```
Matrix A(3,4), B(4,6), C;
ProMatrix D(4);
C = A * B; // legal operation
C = B * A; // illegal operation
C = A * D; // coercion:  converts D
```

The method first verifies whether the operation is legal. If it is not a zero matrix is returned else a new matrix is allocated and the multiplications are carried out.
Multiplication or division of a matrix and vector by a factor are also defined through operators. They perform a scalar operation on each of the element in the array. A vector may e.g. be multiplied by a factor,

——— *scalar operator* ———

Implementation:

```
Vector& Vector::operator *= (double scal)
{
  for (i=1; i<=no_row; i++)
    v[i] *= scal;
  return *this;
}
```

Syntax:

```
double x=3.5;
Vector a(4);
a *= x;
```

A.2.3 Input and output operators

The input and output of vectors and matrices can be performed by the stream operators. A stream is a sequence of characters consisting of e.g. integers, floating point variables or character strings. The stream input operator, `>>`, and the stream output operator, `<<`, can be overloaded to apply to a specific data type. For vectors and matrices the output operator sets up object body for output, e.g.

——— *output* ———

Implementation:

```
ostream& operator(ostream& os, Vector& b)
{
  os.precision(4);
  os.setf(ios::showpoint);
  for (i=1; i<=no_row; i++)
    os << v[i] << endl;
  return os;
}
```

Syntax:

```
Vector a(2);
cout << "a = " << endl << a;
```

———

The stream output operator, `<<`, is used for generating a stream consisting of a string, `"a = "`, a newline character `endl` and the vector `a` written in a column. The stream is finally send to standard output - usually the screen - through `cout`, which is an object of the output stream class, `ostream`. The operator is declared as a friend method and can therefore access the private attributes directly. The variable, `os`, is an object of the class `ostream`. Among its attributes are flags that defines the output format e.g. the number of decimals on the floating point output. These flags are e.g. set by the member methods, `precision` and `setf`. A detailed description of the stream classes including file handling is e.g. found in Lippman (1989).

A.3 Solution of linear equation systems

An important step in solving many problems by the finite element method is the solution of the global equations. These equations have the form

$$\mathbf{A}\mathbf{x} = \mathbf{b} \tag{A.1}$$

where $\mathbf{A}$ is a real, $n \times n$ matrix. $\mathbf{x}^T = (x_1, ..., x_n)$ is a vector containing the system degrees of freedom - the displacements - and $\mathbf{b}^T = (b_1, ..., b_n)$ is the load vector.
The solution can be obtained by multiplying the load vector by the inverse of the system matrix, i.e. $\mathbf{x} = \mathbf{A}^{-1}\,\mathbf{b}$. The computation of the inverse is, however, expensive and for finite element problems it is often chosen first to factorize the system matrix and then obtain the solution by forward and back substitutions. For general un-symmetric matrices the $\mathbf{LU}$-factorization can be used. Assuming the matrix, $\mathbf{A}$, can be decomposed in an upper triangular matrix, $\mathbf{U}$, and a lower triangular matrix, $\mathbf{L}$, such that

$$\mathbf{A}\mathbf{x} = \mathbf{L}\mathbf{U}\,\mathbf{x} = \mathbf{b} \tag{A.2}$$

The solution is then obtained by solving two new equation systems,

$$\mathbf{L}\,\mathbf{y} = \mathbf{b} \tag{A.3}$$
$$\mathbf{U}\,\mathbf{x} = \mathbf{y} \tag{A.4}$$

Due to the triangular form of the system matrices, the solutions can be obtained directly by substitution.
In finite element problems where the energy functional is symmetric, e.g. potential problems and linear elasticity, the system matrix - the stiffness matrix - is symmetric, positive definite and well-conditioned. These three characteristics imply that a symmetric factorization scheme without pivotation can be used. In the following the $\mathbf{LDL}^T$ scheme for symmetric profile matrices is described in detail, Hededal & Krenk (1993). Still, the development of a simple $\mathbf{LU}$ factorization and solution without pivotation follows entirely the same scheme.

A.3.1 Factorization

A symmetric matrix $\mathbf{A}$ can be factored in the form

$$\mathbf{A} = \mathbf{L}\mathbf{D}\mathbf{L}^T \tag{A.5}$$

in which $\mathbf{L}$ is a lower triangular matrix and $\mathbf{D}$ is a diagonal matrix. The computation of $\mathbf{L}$ and $\mathbf{D}$ is found by considering the submatrix $\mathbf{A}_i$,

$$\mathbf{A}_i = \begin{bmatrix} \mathbf{A}_{i-1} & \mathbf{a}_i \\ \mathbf{a}_i^T & c_i \end{bmatrix} \tag{A.6}$$

where $\mathbf{a}_i^T$ is a subvector containing the off-diagonal elements of row i. The factored form of (A.6)

$$\mathbf{A}_i = \mathbf{L}_i\mathbf{D}_i\mathbf{L}_i^T \tag{A.7}$$

contains the lower triangular matrix

$$\mathbf{L}_i = \begin{bmatrix} \mathbf{L}_{i-1} & \mathbf{0} \\ \mathbf{l}_i^T & 1 \end{bmatrix} \tag{A.8}$$

and the diagonal matrix

$$\mathbf{D}_i = \begin{bmatrix} \mathbf{D}_{i-1} & \mathbf{0} \\ \mathbf{0}^T & d_i \end{bmatrix} \tag{A.9}$$

$\mathbf{0}$ is a zero vector. Substituting (A.8) and (A.9) into (A.7) and comparing with (A.6) gives the two equations:

$$d_i + \mathbf{l}_i^T \mathbf{D}_{i-1} \mathbf{l}_i = c_i \tag{A.10}$$

and

$$(\mathbf{L}_{i-1}\mathbf{D}_{i-1})\mathbf{l}_i = \mathbf{L}_{i-1}\mathbf{u}_i = \mathbf{a}_i \tag{A.11}$$

with $\mathbf{u}_i$ defined as

$$\mathbf{u}_i = \mathbf{D}_{i-1}\mathbf{l}_i \tag{A.12}$$

This enables computation of the factors, $[\mathbf{l}_i^T \;\; d_i]$, in step i. It is seen from (A.11) that the vector $[\mathbf{l}_i^T \;\; d_i]$ contains the same number of leading zeros as the vector $[\mathbf{a}_i^T \;\; c_i]$. Thus, the factored matrix has exactly the same structure as the original one and can be stored in the memory already allocated for the original un-factored matrix.

$$\mathbf{A} = \begin{bmatrix} a_{11} & & & & & \\ a_{21} & a_{22} & & & sym & \\ a_{31} & a_{32} & a_{33} & & & \\ & a_{42} & a_{43} & a_{44} & & \\ & & a_{53} & a_{54} & a_{55} & \\ a_{61} & a_{62} & a_{63} & a_{64} & a_{65} & a_{66} \end{bmatrix} \quad , \qquad \mathbf{p} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 2 \\ 0 \end{bmatrix}$$

Figure A.1: Profile matrix

Consider a profile matrix as in Figure A.1. For each row i the matrix is stored in an array $\mathbf{a}_i$ containing the elements a_{ij}. p_i holds the number of leading zeros in row i, hence the remaining elements of row i are indexed from $j = p_i + 1$ to $j = i$. Using an allocation technique that enables this index style, see e.g. Press et al. (1988), and overwriting $[\mathbf{a}_i^T \;\; c_i]$ with $[\mathbf{l}_i^T \;\; d_i]$ leads to the simple algorithm for factoring profile matrices given as Algorithm A.1.

ALGORITHM A.1: FACTORIZATION

```
for (i = 2 to n)
  for (j = p(i)+2 to i–1) do
    for (k = max(p(i),p(j))+1 to j–1) do
      A(i,j) = A(i,j) – A(j,k) * A(i,k)          (u_ij from (A.11))
  for (j = p(i)+1 to i–1) do
    u = A(i,j)                                   (temporary u_ij)
    A(i,j) = A(i,j) / A(j,j)                     (l_ij from (A.12))
    A(i,i) = A(i,i) – A(i,j) * u                 (d_ii from (A.10))
```

A.3.2 Solution

The solution part has three phases

$$\mathbf{Lz} = \mathbf{b} \tag{A.13}$$

$$\mathbf{Dy} = \mathbf{z} \tag{A.14}$$

$$\mathbf{L}^T\mathbf{x} = \mathbf{y} \tag{A.15}$$

These operations are straight forward if we consider full lower triangular matrices. For profile matrices care must be taken to avoid operating on elements outside the profile in the factored matrix. The forward substitution (A.13) operates on rows from $i = 1$ to n with the elements a_{ij} indexed from $j = p_i + 1$ to i. Dividing with the diagonal terms (A.14) is trivial. The back substitution (A.15) is slightly modified. Instead of performing row operations the solution is obtained by gradually modifying the right hand side. Rearranging the subsystem

$$\begin{bmatrix} \mathbf{L}_{i-1}^T & \mathbf{l}_i \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} \mathbf{x}_{i-1} \\ x_i \end{bmatrix} = \begin{bmatrix} \mathbf{y}_{i-1} \\ y_i \end{bmatrix} \tag{A.16}$$

yields

$$\begin{bmatrix} \mathbf{L}_{i-1}^T & \mathbf{0} \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} \mathbf{x}_{i-1} \\ x_i \end{bmatrix} = \begin{bmatrix} \mathbf{y}_{i-1} - \mathbf{l}_i x_i \\ y_i \end{bmatrix} \tag{A.17}$$

whereby the solution emerges as $\mathbf{x}$. The total solution scheme is given as Algorithm A.2.

A.3.3 Constrained systems

Algorithm A.1 and Algorithm A.2 are formulated for equation systems without prescribed displacements where the vector $\mathbf{x}$ contains all the unknown values, while $\mathbf{b}$ holds the known values. The output of the solution function is the unknown $\mathbf{x}$. For constrained systems the vector $\mathbf{x}$ contains some known values and the corresponding elements in $\mathbf{b}$ are unknown. The solution function should therefore be modified to supply the unknown values in both $\mathbf{x}$ and $\mathbf{b}$, while the known values are left untouched in the two vectors.

ALGORITHM A.2: PROFILE SOLVE

```
for (i = 2 to n) do
  for (j = p(i)+1 to i-1) do
    x(i) = x(i) - A(i,j) * x(j)                    (z_i from (A.13))
for (i = 1 to n) do
  x(i) = x(i) / A(i,i)                             (y_i from (A.14))
for (i = n downto 2) do
  for (j = p(i)+1 to i-1) do
    x(j) = x(j) - A(i,j) * x(i)                    (x_i from (A.15))
```

The equation system, (A.1), can be divided into a part that relates to the free displacements, $\mathbf{x}_f$, and a part relating to the prescribed displacements, $\mathbf{x}_c$. The load vector is similarly divided in two: $\mathbf{b}_f$ corresponding to the free terms and a part related to the prescribed displacements, $\mathbf{b}_c + \mathbf{b}_c^0$. $\mathbf{b}_c$ are the unknown reactions, while $\mathbf{b}_c^0$ contains the contribution from a prescribed load. The constrained equation system can be written as

$$\begin{bmatrix} \mathbf{A}_{ff} & \mathbf{A}_{fc} \\ \mathbf{A}_{fc} & \mathbf{A}_{cc} \end{bmatrix} \begin{bmatrix} \mathbf{x}_f \\ \mathbf{x}_c \end{bmatrix} = \begin{bmatrix} \mathbf{b}_f \\ \mathbf{b}_c + \mathbf{b}_c^0 \end{bmatrix} \tag{A.18}$$

The complete solution to the system is available once the free displacements have been determined, thus only the free part of the system matrix, $\mathbf{A}_{ff}$, needs to be factorized. Usually the equations are not ordered in free and prescribed terms, but mix arbitrarily. Instead of reordering the equation system the state of a displacement is identified by a boolean array, $\mathbf{f}$, i.e. $f_i = 1$ for prescribed displacement and $f_i = 0$ for unknown displacement.
During factorization terms related to the prescribed displacements, $\mathbf{A}_{fc}$ and $\mathbf{A}_{cc}$, are not active and must be left unchanged. These terms are identified by the boolean array, $\mathbf{f}$, and are thus omitted by checking for constraints before entering each loop in Algorithm A.1 and Algorithm A.2.
Prescribed displacements impose loads on the free part of the system that must be taken into account during solution. The free part of the load vector are modified with the contribution from the prescribed displacements,

$$\mathbf{b}_f = \mathbf{b}_f - \mathbf{A}_{fc}\,\mathbf{x}_c \tag{A.19}$$

The evaluation of (A.19) is carried out most effectively by accessing the lower part of the matrix $\mathbf{A}$ row-wise as indicated in Algorithm A.3.
After having solved the free equation system calculation of the reactions, $\mathbf{b}_c$, remains. The reactions receives contributions from the prescribed load, $\mathbf{b}_c^0$, and a load imposed by the deformation of the system, $\mathbf{A}_{fc}\,\mathbf{x}_f$, i.e.

$$\mathbf{b}_c = -\mathbf{b}_c^0 + \mathbf{A}_{fc}\,\mathbf{x}_f \tag{A.20}$$

In this operation all elements related to prescribed displacements are used. As for the modification of the load vector the lower part of $\mathbf{A}$ will be considered row-wise seaking the active elements as it is illustrated in Algorithm A.4.

Algorithm A.3: Load vector modification

```
for (i = 1 to n) do
  for (j = p(i)+1 to i–1) do
    if (f(i) = 1 and f(j) = 0)
      b(j) = b(j) – A(i,j) * x(i)
    else if (f(j) = 1 and f(i) = 0)
      b(i) = b(i) – A(i,j) * x(j)
```

Algorithm A.4: Calculation of reactions

```
for (i = 1 to n) do
  if (f(i) = 1)
    b(i) = –b(i)
for (i = 1 to n) do
  for (j = p(i)+1 to i–1) do
    if (f(i) = 1)
      b(i) = b(i) + A(i,j) * x(j)
    if (f(j) = 1 and j ≠ i)
      b(j) = b(j) + A(i,j) * x(i)
```

Implementation with the algebraic classes

The solution of linear equations is highly related to the structure of the system matrix and it is therefore natural to implement solution methods in conjunction with the Matrix and ProMatrix classes. The full Matrix uses a simple **LU** scheme without pivotation, while the ProMatrix class implements the $\mathbf{LDL}^T$ scheme described in this section. Both classes contain methods factorizing and solving unconstrained as well as constrained systems.

unconstrained system

Syntax:

```
Matrix A(n,n);
Vector b(n);
factor(A);
solve(A,b);
```

The method `factor` replaces the original matrix by its factored version. The `solve` method takes the factored matrix and the load vector as input. The solution is stored in the load vector. For constrained systems it is necessary to mark the terms related to the prescribed displacements. The boolean array `f` marks the free and prescribed displacements. This array is passed along with the system matrix to the `factor` and the `solve` method.

constrained system

Syntax:

```
Matrix A(n,n);
Vector b(n), x(n);
IntArray f(n);
factor(A,f);
solve(A,x,b,f);
```

Syntax:

```
ProMatrix A(n);
Vector b(n), x(n);
IntArray f(n);
factor(A,f);
solve(A,x,b,f);
```

For constrained system solution consists of determining unknown components in both the displacement vector, `x`, and the load vector, `b`. Therefore the `solve` method takes both these vectors as arguments and returns them with their unknown values filled in.
The names of the two solution methods, `factor` and `solve`, are shared by all versions of the methods. same name is used for all types of problems and matrices. Due to polymorphism, the program is able to decide which version to evoke by looking at the arguments. This facility e.g. enables the programmer to change the matrix type without having to alter the entire code.
The methods are declared as friends to the Vector class and either the Matrix or ProMatrix class. The methods can thus access the private members directly, which is vital for the efficiency of the methods.

Table A.1: Declaration of Vector class

	Attributes:	
`int`	`no_row`	No. of rows
`double`	`*v`	1D floating point array
	Constructors & destructor:	
	`Vector()`	Default: Set all attributes to 0
	`Vector(int nr)`	Allocate vector
	`Vector(Vector& b)`	Copy: Initialize object as a copy of `b`
	`~Vector()`	Destructor: Free allocated storage
	Assignment operators:	
`Vector&`	`= (Vector& b)`	Assign `b` to object
`Vector&`	`= (double scal)`	Assign `scal` to all members
`double&`	`() (int i)`	Assign/access member
`double*`	`pointer()`	Return array pointer
	Vector–Vector operators:	
`Vector`	`- ()`	Unary minus: $-\mathbf{b}$
`Vector`	`+ (Vector& b)`	Addition of two vectors
`Vector&`	`+= (Vector& b)`	Addition with assign
`Vector`	`- (Vector& b)`	Substraction of two vectors
`Vector&`	`-= (Vector& b)`	Substraction with assign
	Vector–Scalar operators:	
`Vector`	`* (double scal)`	Multiplication with scalar
`Vector&`	`*= (double scal)`	Scalar multiplication with assign
`Vector`	`/ (double scal)`	Division with scalar
`Vector&`	`/= (double scal)`	Scalar division with assign
	Vector products - friend methods:	
`double`	`dot(Vector& b, Vector& c)`	Scalar product
`Vector`	`cross(Vector& b, Vector& c)`	Cross product
`Matrix`	`* (Vector& b, Vector& c)`	Exterior product
`double`	`length(Vector& b)`	Euclidian norm
`double`	`norm(Vector& b, int n)`	`n`-norm
	Solve methods - friend methods:	
`char`	`solve(Matrix& A, Vector& b)`	Solve un-symmetric linear system
`char`	`solve(Matrix& A, Vector& x, Vector& b, IntArray& f)`	Solve un-symmetric constrained linear system
`char`	`solve(ProMatrix& A, Vector& b)`	Solve symmetric linear system
`char`	`solve(ProMatrix& A, Vector& x, Vector& b, IntArray& f)`	Solve symmetric constrained linear system
	Input & output operators - friend methods:	
`istream&`	`>> (istream& is, Vector& b)`	Read formatted input
`ostream&`	`<< (ostream& os, Vector& b)`	Write formatted output
	Miscellaneous:	
`int`	`size(int i)`	Dimension of vector

Table A.2: Declaration of Matrix class

	Attributes:	
`int`	`no_row`	No. of rows
`int`	`no_col`	No. of columns
`double`	`**m`	2D floating point array
	Constructors & destructor:	
	`Matrix()`	Default: Set all attributes to 0
	`Matrix(int nr, int nc)`	Allocate matrix
	`Matrix(Matrix& A)`	Copy: Initialize object as a copy of `A`
	`~Matrix()`	Destructor: Free allocated storage
	Assignment operators:	
`Matrix&`	`= (Matrix& A)`	Assign `A` to object
`Matrix&`	`= (double scal)`	Assign `scal` to all members
`double&`	`() (int i, int j)`	Assign/access member
`double**`	`pointer()`	Return array pointer
	Matrix–Matrix operators:	
`Matrix`	`- ()`	Unary minus: $-\mathbf{A}$
`Matrix`	`+ (Matrix& A)`	Addition of two matrices
`Matrix&`	`+= (Matrix& A)`	Addition with assign
`Matrix`	`- (Matrix& A)`	Substraction of two matrices
`Matrix&`	`-= (Matrix& A)`	Substraction with assign
`Matrix`	`* (Matrix& A, Matrix& B)`	Multiplication of two matrices
`Matrix&`	`*= (Matrix& A)`	Multiplication with assign
	Matrix–Scalar operators:	
`Matrix`	`* (double scal)`	Multiplication with scalar
`Matrix&`	`*= (double scal)`	Scalar multiplication with assign
`Matrix`	`/ (double scal)`	Division with scalar
`Matrix&`	`/= (double scal)`	Scalar division with assign
	Matrix–Vector operators - friend methods:	
`Vector&`	`* (Matrix& A, Vector& b)`	Multiply matrix and vector
`Vector&`	`* (Vector& b, Matrix& A)`	Multiply vector and matrix
	Factor & solve methods - friend methods:	
`char`	`factor(Matrix& A)`	**LU** factors of matrix
`char`	`solve(Matrix& A, Vector& b)`	Solve linear system
`char`	`factor(Matrix& A, IntArray& f)`	**LU** factors of constrained matrix
`char`	`solve(Matrix& A, Vector& x, Vector& b, IntArray& f)`	Solve constrained linear system
`double`	`det(Matrix A)`	Determinant
`Matrix`	`inv(Matrix A)`	Inverse of matrix
	Input & output operators - friend methods:	
`istream&`	`>> (istream& is, Matrix& A)`	Read formatted input
`ostream&`	`<< (ostream& os, Matrix& A)`	Write formatted output
	Miscellaneous:	
`int`	`size(int i)`	Dimension of matrix
`Matrix`	`T()`	Transpose of matrix

Table A.3: Declaration of ProMatrix class

	Attributes:	
`int`	`no_row`	No. of rows
`int`	`no_col`	No. of columns
`double`	`**m`	2D floating point array
`IntArray`	`p`	Profile array
	Constructors & destructor:	
	`ProMatrix()`	Default: Set all attributes to 0
	`ProMatrix(int nr)`	Allocate lower triangular matrix
	`ProMatrix(int nr, IntArray& p)`	Allocate profile matrix
	`ProMatrix(ProMatrix& A)`	Copy: Initialize object as a copy of `A`
	`~ProMatrix()`	Destructor: Free allocated storage
	Assignment operators:	
`ProMatrix&`	`= (ProMatrix& A)`	Assign `A` to object
`ProMatrix&`	`= (double scal)`	Assign `scal` to all members
`double&`	`() (int i, int j)`	Assign/access member
`double**`	`pointer()`	Return array pointer
	Coercion operator:	
	`Matrix()`	Force conversion to Matrix
	ProMatrix–ProMatrix operators:	
`ProMatrix`	`- ()`	Unary minus: $-\mathbf{A}$
`ProMatrix`	`+ (ProMatrix& A)`	Addition of two matrices
`ProMatrix&`	`+= (ProMatrix& A)`	Addition with assign
`ProMatrix`	`- (ProMatrix& A)`	Substraction of two matrices
`ProMatrix&`	`-= (ProMatrix& A)`	Substraction with assign
	ProMatrix–Scalar operators:	
`ProMatrix`	`* (double scal)`	Multiplication with scalar
`ProMatrix&`	`*= (double scal)`	Scalar multiplication with assign
`ProMatrix`	`/ (double scal)`	Division with scalar
`ProMatrix&`	`/= (double scal)`	Scalar division with assign
	ProMatrix–Vector operators - friend methods:	
`Vector&`	`* (ProMatrix& A, Vector& b)`	Multiply matrix and vector
`Vector&`	`* (Vector& b, ProMatrix& A)`	Multiply vector and matrix
	Factor & solve methods - friend methods:	
`char`	`factor(Matrix& A)`	$\mathbf{LDL}^T$ factors of matrix
`char`	`solve(ProMatrix& A, Vector& b)`	Solve linear system
`char`	`factor(ProMatrix& A, IntArray& f)`	$\mathbf{LDL}^T$ factors of constrained matrix
`char`	`solve(ProMatrix& A, Vector& x, Vector& b, IntArray& f)`	Solve constrained linear system
`double`	`det(ProMatrix A)`	Determinant
	Input & output operators - friend methods:	
`istream&`	`>> (istream& is, ProMatrix& A)`	Read formatted input
`ostream&`	`<< (ostream& os, ProMatrix& A)`	Write formatted output
	Miscellaneous:	
`int`	`size(int i)`	Dimension of matrix

Table A.4: Declaration of IntArray class

	Attributes:	
`int`	`no_row`	No. of rows
`int&`	`*v`	1D integer array
	Constructors & destructor:	
	`IntArray()`	Default: Set all attributes to 0
	`IntArray(int nr)`	Allocate array
	`IntArray(IntArray& b)`	Copy: Initialize object as a copy of `b`
	`~IntArray()`	Destructor: Free allocated storage
	Assignment operators:	
`IntArray&`	`= (IntArray& b)`	Assign `b` to object
`IntArray&`	`= (int scal)`	Assign `scal` to all members
`int`	`() (int i)`	Assign/access member
`int*`	`pointer()`	Return array pointer
	Input & output operators - friend methods:	
`istream&`	`>> (istream& is, IntArray& b)`	Read formatted input
`ostream&`	`<< (ostream& os, IntArray& b)`	Write formatted output
	Miscellaneous:	
`int`	`size(int i)`	Dimension of array

Table A.5: Declaration of Int2DArray class

	Attributes:	
`int`	`no_row`	No. of rows
`int`	`no_col`	No. of columns
`int`	`**m`	2D integer array
	Constructors & destructor:	
	`Int2DArray()`	Default: Set all attributes to 0
	`Int2DArray(int nr, int nc)`	Allocate array
	`Int2DArray(Int2DArray& A)`	Copy: Initialize object as a copy of `A`
	`~Int2DArray()`	Destructor: Free allocated storage
	Assignment operators:	
`Int2DArray&`	`= (Int2DArray& A)`	Assign `A` to object
`Int2DArray&`	`= (int scal)`	Assign `scal` to all members
`int`	`() (int i, int j)`	Assign/access member
`int**`	`pointer()`	Return array pointer
	Input & output operators - friend methods:	
`istream&`	`>> (istream& is, Int2DArray& A)`	Read formatted input
`ostream&`	`<< (ostream& os, Int2DArray& A)`	Write formatted output
	Miscellaneous:	
`int`	`size(int i)`	Dimension of array

A.4 Examples

This section consists of five small C++ programs that use the algebraic classes for linear algebra and vector calculus. In each example the full C++ code is given assuming that the algebraic classes are available to be linked along with the main program. The declarations of the algebraic classes used in the examples are assumed to be contained in four header files: `vector.h`, `matrix.h`, `promat.h` and `intarr.h`. Test input and the corresponding output conclude each example.

Example 1: Vector calculus

This example takes two vectors, `a` and `b`, of the same dimension, `n`, as input from the screen. The vector product methods are used for evaluating the scalar product, the length of a vector and the cross product. The area of a triangle described by the two vectors is then found as $A = 0.5|\mathbf{a} \times \mathbf{b}|$. The vectors are declared by the default constructor, `Vector()`, and later allocated using the constructor `Vector(int nr)`, which sets the size of the vectors.

```
// File: ex1.c
// Vector calculus

#include <iostream.h>
#include "vector.h"

void main()
{
  int n;                            // declare variables
  Vector a,b,c;

  cin >> n;                         // read dimension of system

  a = Vector(n);                    // allocate vectors
  b = Vector(n);

  cin >> a >> b;                    // read formatted vector input
  cout << "Vector 'a' = " << a;     // echo input
  cout << "Vector 'b' = " << b;

  cout << "Scalar product = " << dot(a,b) << endl;
  cout << "Length of 'a'  = " << length(a) << endl;

  c = cross(a,b);
  cout << "Cross product = " << c;
  cout << "Area of triangle = " 0.5*length(c) << endl;
}
```

```
INPUT:
  3                                          <-- dimension
  1 1 0                                      <-- vector a
  0 1 0                                      <-- vector b

OUTPUT:
  Vector 'a' =
     1.0000
     1.0000
     0.0000

  Vector 'b' =
     0.0000
     1.0000
     0.0000

  Scalar product = 1.0000

  Length of 'a' = 1.4142

  Cross product =
     0.0000
     0.0000
     1.0000

  Area of triangle = 0.5000
```

Example 2: Matrix operations

This example illustrates different matrix operations. Two fully populated matrices are read and echoed using the overloaded stream operators, `>>` and `<<`. The transpose of `A` is found using the member function, `T()`. The matrix `C` is declared by the default constructor, `Matrix()`, thus no memory has yet been allocated. The operation `3*A` creates a new `Matrix`, which is then assigned to `C`, replacing the original empty matrix. The matrix multiplication, `A*B`, is carried out if the dimensions of `A` and `B` match. For square matrices it is possible to evaluate the determinant and the inverse. Comparing the dimensions, `no_row = size(1)` and `no_col = size(2)`, decides whether these are evaluated. If so, the determinant is computed and the matrix `C` is redefined as the inverse of `A`. The `inv` method takes a copy of `A` as argument, thus does not affect the global value of `A`. The operation `A*C` verifies that the inversion is correct.

```
// File: ex2.c
// Matrix operations

#include <iostream.h>
#include "matrix.h"

void main()
{
  int nr,nc;                          // declare variables
  Matrix A,B,C;

  cin >> nr >> nc;                    // dimension of A
  A = Matrix(nr,nc);                  // allocate matrix A
  cin >> A;                           // read A

  cin >> nr >> nc;                    // dimension of B
  B = Matrix(nr,nc);                  // allocate matrix B
  cin >> B;                           // read B

  cout << "A^T = " << A.T();          // transpose of A
  C = 3*A;                            // create C = 3A
  cout << "A+C = " << A + C;          // add A and C
  cout << "A/3 = " << A/3;            // divide A by 3
  cout << "A*B = " << A*B;            // multiply A and B

  if (A.size(1) == A.size(2))         // square matrix?
  {
    cout << "Det(A) = " << det(A);    // determinant
    C = inv(A);                       // redefine C as inverse of A
    cout << "A*inv(A) = " << A*C;     // identity matrix!
  }
}
```

```
INPUT:
  3 3                                          <-- dimensions of A
  6 5 4                                        <-- matrix A
  1 4 3
  2 -1 4
  3 2                                          <-- dimensions of B
  1 2                                          <-- matrix B
  2 3
  3 4

OUTPUT:
  A^T =
    6.0000      1.0000      2.0000
    5.0000      4.0000     -1.0000
    4.0000      3.0000      4.0000

  A+C =
   24.0000     20.0000     16.0000
    4.0000     16.0000     12.0000
    8.0000     -4.0000     16.0000

  A/3 =
    2.0000      1.6667      1.3333
    0.3333      1.3333      1.0000
    0.6667     -0.3333      1.3333

  A*B =
   28.0000     43.0000
   18.0000     26.0000
   12.0000     17.0000

  Det(A) = 88.0000

  A*inv(A) =
      1.0000 -2.2204e-16      0.0000
  2.7756e-17      1.0000  2.7756e-17
  5.5511e-17 -2.2204e-16      1.0000
```

Example 3: ProMatrix operations

The operations of the symmetric profile matrix class, `ProMatrix`, are the same as for `Matrix`. The allocation of a profile matrix requires an array of leading zeroes, `p`, to be defined before the matrices, `A` and `B`, are allocated and read. The addition and substraction of symmetric matrices result in new symmetric matrices, whereas multiplication generally gives un-symmetric, fully populated matrices. The multiplication of operation, `A*B`, first uses the coercion operator, `Promatrix::Matrix()` and then calls the `Matrix` operator `*`. In the addition operation the `ProMatrix` is converted to a `Matrix` before the `Matrix` operator `+` is called.

```
// File: ex3.c
// ProMatrix operations

#include <iostream.h>
#include "matrix.h"
#include "promat.h"
#include "intarr.h"

void main()
{
  int n;                          // declare variables
  ProMatrix A,B;
  IntArray p;
  Matrix C;

  cin >> n;                       // read dimension of system

  p = IntArray(n);                // allocate integer array
  cin >> p;                       // read profile of A
  A = ProMatrix(n,p);             // allocate A
  cin >> A;                       // read formatted matrix input

  cin >> p;                       // read profile of B
  B = ProMatrix(n,p);             // allocate B
  cin >> B;                       // read formatted matrix input

  cout << "A+B = " << A+B;        // addition ==> ProMatrix
  cout << "A-B = " << A-B;        // substraction ==> ProMatrix

  cout << "A*B = " << A*B;        // multiplication ==> Matrix

  C = Matrix(n,n);                // allocate full Matrix
  cin >> C;                       // read Matrix input
  cout << "C+A = " << C+A;        // Matrix + ProMatrix ==> Matrix
}
```

```
INPUT:
  4                                          <-- dimension
  0 0 1 2                                    <-- profile of A
  1                                          <-- profile matrix A
  2 3
    3 2
      5 1
  0 1 0 1                                    <-- profile of B
  3                                          <-- profile matrix B
    2
  6 2 3
    8 3 4
  1 3 4 5                                    <-- matrix C
  4 2 3 4
  3 4 5 7
  5 3 2 4

OUTPUT:
  A+B =
    4.0000
    2.0000      5.0000
    6.0000      5.0000      5.0000
       ---      8.0000      8.0000      5.0000

  A-B =
   -2.0000
    2.0000      1.0000
   -6.0000      1.0000     -1.0000
       ---     -8.0000      2.0000     -3.0000

  A*B =
    3.0000      4.0000     10.0000     16.0000
   24.0000     12.0000     27.0000     33.0000
   12.0000     50.0000     27.0000     50.0000
   30.0000     18.0000     18.0000     19.0000

  C+A =
    2.0000      5.0000      4.0000      5.0000
    6.0000      5.0000      6.0000      4.0000
    3.0000      7.0000      7.0000     12.0000
    5.0000      3.0000      7.0000      5.0000
```

Example 4: Solution of linear equation systems

A central part of linear algebra is the solution linear equation systems on the form,

$$\mathbf{Ax} = \mathbf{b}$$

where system matrix, $\mathbf{A}$, is an n-dimensional square matrix and $\mathbf{x}$ and $\mathbf{b}$ are the solution vector and the load vector, respectively. The solution, $\mathbf{x}$, is found by multiplying the specified load vector, $\mathbf{b}$, with the inverse of the system matrix, i.e.

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$$

The program makes use of the method `inv` to evaluate the inverse of the system matrix. The operation `inv(A)*b` creates a new vector, which is assigned to the solution vector, `x`.

```
// File: ex4.c
// Solution of un-symmetric linear equations

#include <iostream.h>
#include "vector.h"
#include "matrix.h"

void main()
{
  int i,j,n;                        // declare variables
  Matrix A;
  Vector b,x;

  cin >> n;                         // read dimension of system

  A = Matrix(n,n);                  // allocate memory
  b = Vector(n);

  cin >> A >> b;                    // read formatted input
  cout << "System matrix = " << A;  // echo input
  cout << "Load vector = " << b;    // echo input

  x = inv(A) * b;                   // solve linear equations
  cout << "Solution = " << x;       // print solution vector
}
```

```
INPUT:
  3                                          <-- dimension
  6 5 4                                      <-- system matrix A
  1 4 3
  2 -1 4
  28 18 12                                   <-- load vector b

OUTPUT:
  System matrix =
    6.0000      5.0000      4.0000
    1.0000      4.0000      3.0000
    2.0000     -1.0000      4.0000

  Load vector =
   28.0000
   18.0000
   12.0000

  Solution =
    1.0000
    2.0000
    3.0000
```

Example 5: Solution of finite element equations

The ProMatrix class provides two methods, factor and solve, that enables solution of a constrained system. An array, `f`, is introduced to mark the prescribed displacements – `f(i) = 1` for prescribed displacement and `f(i) = 0` if the displacement component is free. This array is passed to the methods along with the stiffness matrix, `K`, the displacement vector, `x`, and the load vector, `b`.

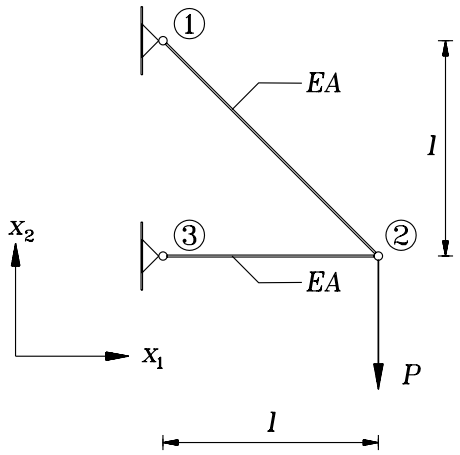


Figure A.2: Two-bar truss

Consider the plane truss structure in Figure A.2. It consists of two linear bar elements with an axial stiffness of EA. The global stiffness matrix is found by assembling the two element contributions, see e.g. Section 7.2.

$$\mathbf{K} = \frac{EA}{l}\begin{bmatrix} \frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & 0 & 0 \\ -\frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} & 0 & 0 \\ -\frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2}+1 & -\frac{\sqrt{2}}{2} & -1 & 0 \\ \frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & 0 & 0 \\ 0 & 0 & -1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

The structure is loaded by a vertical point load, P, in node 2 and fixed in both displacement components at node 1 and 3, whereby the load vector, $\mathbf{b}$, and the array, $\mathbf{f}$, become

$$\mathbf{b}^T = [\,0 \quad 0 \quad 0 \quad P \quad 0 \quad 0\,] \quad , \quad \mathbf{f} = [\,1 \quad 1 \quad 0 \quad 0 \quad 1 \quad 1\,]$$

The following example uses the axial stiffness, $EA = 1$, the length, $l = 1$ and the load $P = -1$. The program first defines the stiffness matrix as a profile matrix, stores the load vector in a vector and fixity is stored as an integer array. The factorization is carried out omitting the rows and columns corresponding to a prescribed zero displacement, `f(i) = 1`. Next, the unknown parts of the displacements and loads are found using the `solve` method.

```
// File: ex5.c
// Solution of finite element equations

#include <iostream.h>
#include "vector.h"
#include "promat.h"
#include "intarr.h"

void main()
{
  int n;                          // declare variables
  Vector b,x;
  ProMatrix A;
  IntArray p,f;

  cin >> n;                       // read size of system

  p = IntArray(n);                // allocate integer array
  cin >> p;                       // read profile array

  K = ProMatrix(n,p);             // allocate profile matrix
  cin >> K;                       // read stiffness matrix
  cout << "K = " << K;            // echo stiffness matrix

  b = Vector(n);                  // allocate vectors
  x = Vector(n);
  cin >> b;                       // read load vector

  f = IntArray(n);                // allocate fix array
  cin >> f;                       // read fix array

  factor(K,f);                    // LDL^T factors of K
  cout << "LD = " << K;           // print matrix factors
  solve(K,x,b,f);                 // solve linear equations

  cout << "Displacements = " << x;  // print solution vector
  cout << "Loads = " << b;          // print solution vector
}
```

```
INPUT:
  6                                          <-- problem size
  0 0 0 0 2 5                                <-- profile p
  0.3536                                     <-- stiffness matrix K
 -0.3536      0.3536
 -0.3536      0.3536      1.3536
  0.3536     -0.3536     -0.3536      0.3536
 -1.0000      0.0000      1.0000
  0.0000
  0  0  0 -1  0  0                           <-- load vector b
  1  1  0  0  1  1                           <-- fix array f

OUTPUT:
  K =
    0.3536
   -0.3536      0.3536
   -0.3536      0.3536      1.3536
    0.3536     -0.3536     -0.3536      0.3536
       ---         ---     -1.0000      0.0000      1.0000
       ---         ---         ---         ---         ---      0.0000

  LD =
    0.3536
   -0.3536      0.3536
   -0.3536      0.3536      1.3536
    0.3536     -0.3536     -0.2612      0.2612
       ---         ---     -1.0000      0.0000      1.0000
       ---         ---         ---         ---         -nnn--      0.0000

  Displacements =
    0.0000
    0.0000
   -1.0000
   -3.8281
    0.0000
    0.0000

  Loads =
   -1.0000
    1.0000
    0.0000
   -1.0000
    1.0000
    0.0000
```

A.5 References

Hededal, O. (1993): Finite element with C++ classes. *Proc. 6th Nordic Seminar on Computational Mechanics.* Linköping, Sweden, 1993.

Hededal, O. and Krenk, S. (1993): A Profile Solver in C for Finite Element Equations. *Engineering Mechanics Papers.* No. 13, Dept. Building Technology and Structural Engineering, Aalborg University, Aalborg, Denmark. (to appear in *Computers & Structures*)

Kernighan, B. and Ritchie, D.M. (1991): *The C Programming Language, 2nd Edn.* Prentice Hall, New Jersey, USA, 1991.

Lippman, S.B. (1989): *C++ Primer.* Addison-Wesley, Massachusetts, USA, 1990.

Nielsen, L.O. (1993): A C++ basis for computational mechanics software. Technical University of Denmark, Copenhagen, Denmark, 1993.

Press, W.H., Flannery, B.P., Teukolsky, S.A. and Vetterling, W.T. (1988): *Numerical Recipes in C.* Cambridge University Press, Cambridge, U.K., 1988.

Stroustrup, B. (1991): *The C++ Programming Language, 2nd Edn.* Addison-Wesley, Massachusetts, USA, 1991.

Winder, R. (1991): *Developing C++ Software.* John Wiley & Sons Ltd, Chichester, U.K., 1991.

Appendix B

Summary

The aim of this thesis has been to investigate the possibilities of using object-oriented programming for finite element programming. Chapter 1 considers the requirements for a flexible framework for finite element programming motivating the use of object-oriented programming. A brief introduction to the object-oriented concepts, e.g. objects, inheritance and polymorhism, and their representation in C++ is given. The chapter is concluded by a review of central papers on object-oriented finite elements. Then follows the two main parts of the thesis: In Chapter 2-5 the finite element formulations of linear potential problems and linear elasticity are considered and a framework for programming finite elements is established. In Chapter 6-8 the linear framework is extended to deal with non-linear problems. First non-linear solution methods are introduced and then different non-linear finite element formulations are presented. Chapter 9 gives the conclusions.
In Chapter 1 the requirements for a flexible framework for finite element programming are considered motivating the use of object-oriented programming. The numerical requirements concern efficiency and robustness of the algorithms, and the possibility of using different elements, materials or solution algorithms. Furthermore, the structure of a finite element code should enable the program to be specialized and expanded without increasing the complexity of the code. Structuring the program in objects gives a distributed architecture with very few bindings between different parts of the program. It is thus possible to introduce new facilities without affecting the existing part of the program. Furthermore, in object-oriented programming the tedious parts of the code, e.g. the input/output facilities, can be inherited from an existing object, hence the programmer can concentrate on formulating the problem dependent part.
Chapter 2-5 describe the development of an object-oriented framework for finite element programming, ObjectFEM. In Chapter 2 the finite element formulations of linear potential problems and linear elasticitity theory are considered. The purpose is to identify a general structure that applies to a large number of finite element problems. In mechanics the finite element formulation is usually based on a balance equation, e.g. the Poisson equation for potential problems or static equilibrium for elastic bodies. The balance equation, which defines a generalized divergence operator, is reformulated by taking a weighted mean thus obtaining the weak form used for the finite element approximation. The weak form defines the generalized gradient operator which is adjoint to the divergence. Discretizing the

domain into elements connected at the nodes leads to the finite element approximation. The discretized problem is described in terms of the nodal degrees-of-freedom and the corresponding loads. Within each element the distributed properties are processed into discrete form, i.e. element stiffness and load. The derived results, i.e. strain and stress, are also defined by the element. The element degrees-of-freedom, stiffness matrix and load vector are assembled into a global system of equation containing the complete finite element model. Solving this gives the unknown nodal degrees-of-freedom and loads.
The concepts established in Chapter 2 are used for defining the FEM classes: Node, Element and Material. These define a standard interface consisting of shared methods and FEM methods. The shared methods take care of the model definition and generation, the assembly of and retrieval from the global equation system and input/output operations. By defining a number of problem parameters, e.g. the number of element nodes and the number of degrees-of-freedom, the shared methods can be used directly by the subclasses. The FEM methods specify the finite element formulation, e.g. element stiffness or strain, hence must be defined for each element type or material model. A new element or material is implemented as a subclass of either the Element or Material superclass.
The finite element model is stored in linked lists. The list can be extended gradually, thus it is not necessary to specify the problem size initially. A linked list uses dynamically allocated memory and it is vital that it is manipulated safely, hence a List class is introduced defining methods for adding, removing and searching for items in the list. The class is defined as a template, i.e. a general data type which by typecasting can be used for storing different data types, e.g. Node and Element. The algebraic classes, Vector and Matrix, define a symbolic notation for programming linear algebra. They consist of overloaded operators, which simulate the mathematical notation, and of methods for solution of linear equation systems. The classes are specialized according to the structure, e.g. profile matrices or sparse matrices. The lists, algebraic classes and FEM classes form a macro-language which is used for writing applications. The application is a user-defined module controlling the complete finite element analysis. It consists of model definition and generation, formation and solution of the global equations, and a postprocessing part.
In Chapter 4 and 5 the basic framework, i.e. Element and Material, is customized to specific problems. The linear potential element and the linear elastic solid mechanics element are formulated using the isoparametric element concept. The two isoparametric elements belong to a family of isoparametric continuum elements and a superclass, Continuum, is derived from Element. A Gausspoint class is introduced storing the Gauss coordinates and weights which are used in the numerical integration scheme. Serendipity elements for 3D and 2D potential problems are derived from the Continuum superclass and simple material models are derived from the Material class. Linear elastic solid mechanics elements are implemented by the Solid class. Using the potential elements as superclasses the modification are limited to a redefinition the dimensions of the element matrices. The Material class is specialized to linear isotropic elastic materials to be used by the solid elements, i.e. 3D: Elastic, 2D: PlaneStrain and PlaneStress.
The second part of the thesis, Chapter 6-8, concerns non-linear finite elements. In Chapter 6 the solution of non-linear finite element equations is considered. First, a general introduction to non-linear finite elements in relation to non-linear solution strategies is

given. The solution algorithms are predictor-corrector strategies consisting of an estimate based on a linear tangent relation followed by equilibrium iterations, where the estimate is corrected until equilibrium is obtained. The first estimate is usually based on a tangent stiffness relation. The tangent stiffness is assembled from element contributions. The main component in the correction first estimate is the residual force, i.e. the unbalance between the internal force produced by the deformation of the structure and the external applied load. The internal force vector, which must be formed in every iteration, is assembled of element contributions. It classifies the non-linear problems in two groups. Those where the internal force can be evaluated by explicit integration over the element, e.g. geometrically non-linear problems, and those that require implicit integration, i.e. where the internal force is integrated from a stress state that in each point is first integrated over the complete load history. The extension of the framework to handle non-linear problems thus requires the Element class to be able to evaluate the tangent stiffness and the internal force.

Two iterative solution strategies are presented: the arc-length method and the orthogonal residual method, which is an alternative to the widely used arc-length method. In order to make the different methods into robust codes the passage of load limit points and strategies for controlling the increment size are discussed. An application for non-linear finite element analysis which uses either the arc-length method or the orthogonal residual method is presented.

In Chapter 7 an elastic bar element with finite deformations is presented as an example of non-linear problems with explicit evaluation of the internal force. The bar element uses the Green strain measure to describe the deformation of the bar. A tangent stiffness is established and the internal force is defined in terms of the current displacement state. Neglecting the non-linear terms a simple formulation for a linear bar element emerges. This linear element, Bar, is used as superclass for the non-linear class, NlBar. The chapter is concluded by 2 examples where the equilibrium paths of non-linear truss structures are traced using the non-linear solution algorithms presented in chapter 6.

Chapter 8 presents elasto-plastic material models as an example of non-linear problems where the internal force can not be evaluated explicitly for a given displacement state. The theory for hardening plasticity and the integration of the constitutive relations are shortly resumed. The Element class is modified to handle problems where the material properties are not constant over the element. The Gausspoint class is extended in order to be able to store the current state of stress and strain. A Plastic material class, which serves as superclass for elasto-plastic materials, is derived from Elastic defining the methods necessary to evaluate the tangent stiffness and to integrate the stress used to evaluate the internal force. von Mises plasticity illustrates the implementation of an elasto-plastic model.

The experience with object-oriented programming for linear and non-linear finite elements is discussed in the concluding Chapter 9. It is found that objects can structure the finite element analysis in a highly distributed architecture where the different parts can be developed independently of each other. The finite element formulation is handled by the FEM classes which define a programming framework. The FEM classes enable reuse of the more tedious parts, such as input/output. Thereby the programmer is able to concentrate

on the finite element formulation. The framework can thus be used for fast prototyping of elements, materials and solution methods. The algebraic classes and the applications do not require the programmer to change the programming style dramatically because they in general use a syntax similar to that of procedural programs. However, deriving an element or a material from a base class requires the programmer to be familiar with the framework and basic features in object-oriented programming. Thus a basic knowledge of object-oriented programming is necessary to get the full advantage of an object-oriented framework for finite elements.

Appendix C

Summary in Danish

Formålet med denne afhandling har været at undersøge mulighederne for at anvende objektorienteret programmering i forbindelse med elementmetode. I kapitel 1 opstilles først en række numeriske og strukturelle krav til et fleksibelt programsystem. Dernæst introduceres centrale begreber i objektorienteret programmering, og deres repræsentation i C++ beskrives kort. Kapitlet afsluttes med en kort gennemgang af nogle centrale referencer om objektorienteret elementmetode. Afhandlingen er herefter opdelt i to hoveddele. Første del, kapitlerne 2-5, introducerer elementmetodeformuleringer med hovedvægt på lineære problemer. Et generelt system til programmering af lineær elementmetode opstilles. Dette system specialiseres derefter til konkrete problemer. Anden del, kapitlerne 6-8, beskæftiger sig med ikke-lineær elementmetode. Det vises, at den ikke-lineære formulering kun indebærer få tilføjelser til den lineære standardsystem. Erfaringerne, som er opnået under udviklingen af programsystemet, danner grundlag for konklusionerne præsenteret i kapitel 9.

I kapitel 1 opstilles kravene for et programsystem til programmering af elementmetode. Det kræves, at programsystemet er numerisk effektivt og robust, samt at det er muligt på simpel vis at vælge mellem forskellige elementer, materialer og løsningsstrategier. Dette skal understøttes gennem en programstruktur, som kan specialiseres og udvides uden at øge kompleksiteten af det samlede system. Objekter strukturerer programmet i selvstændige enheder med få interne bindinger. Derved bliver det muligt at udvikle de forskellige dele af programmet uafhængigt af hinanden.

Kapitlerne 2-5 beskriver udviklingen af et programsystem for lineær elementmetode, ObjectFEM. I kapitel 2 formuleres elementmetoden for potentialproblemer og lineær elasticitetsteori. Formålet er at identificere en generel struktur, der er fælles for en række problemer. Indenfor mekanik er grundlaget for elementmetodeformuleringen ofte en balanceligning, f.eks. Poissons ligning, der beskriver potentialproblemer, eller de statiske ligevægtsligninger indenfor elasticitetsteori. Balanceligningen, som definerer en generaliseret divergensoperator, omformes gennem en vægtet middelværdi til den svage form som anvendes i elementmetoden. Denne operation definerer den generaliserede gradientoperator, som er adjungeret til divergensen. En diskretisering af det betragtede volumen i elementer, som er forbundet i knuder, danner grundlag for elementmetodeapproksimationen. Det diskretiserede problem defineres af knudernes frihedsgrader og de dertil svarende

knudelaster. I hvert element omformes de kontinuerte egenskaber til diskret form, d.v.s. elementstivhed og last. Elementet er ligeledes karakteriseret ved den generaliserede tøjning og spænding. Alle elementers frihedsgrader, stivhedsmatricer og lastvektorer samles til et globale ligningsystem, hvis løsning giver de ukendte knudefrihedsgrader og reaktioner.
De begreber, som er identificeret i kapitel 2, danner grundlag for FEM-klasserne: Node, Element og Material. Disse basisklasserne definerer et standardinterface bestående af fællesmetoder og FEM-metoder. Fællesmetoderne håndterer bl.a. definering og generering af modellen, assemblering af det globale ligningssystem samt input/output operationer. Ved at definere nogle problemparametre, f.eks. antallet af elementknuder og antallet af elementfrihedsgrader, er det muligt at anvende fællesmetoderne til alle problemtyper. FEM-metoderne specificerer elementmetodeformuleringen f.eks. i form af stivhed eller tøjning og må derfor defineres for hvert element eller materialemodel. Nye elementer eller materialer implementeres som en underklasse af enten Element eller Material.
Den fulde elementmetodemodel lagres i kædede lister. En liste kan udvides gradvist, og det er derfor ikke nødvendigt at kende modellens størrelse på forhånd. En kædet liste lagrer dynamisk allokerede objekter. Derfor er det vigtigt, at de manipuleres på en sikker måde, hvilket gøres ved at definere listen som en klasse, hvis metoder kan indsætte, fjerne eller finde objekter i listen. Algebraiske klasser muliggør symbolsk programmering af lineær algebra. De to klasser, Vector og Matrix, simulerer den matematiske syntaks ved brug af operatorer, som redefineres så de svarer til deres matematiske modstykke. Da løsningen af lineære ligningssystemer er tæt knyttet til strukturen i systemmatricen, indeholder klasserne ligeledes sådanne løsningsmetoder. De algebraiske klasser, listerne og FEM-klasserne danner et makro-sprog, som anvendes til programmering af applikationer. En applikation programmeres af hver enkelt bruger af systemet og definerer den fulde elementmetodeanalyse bestående af modeldefinition, assemblering og løsning af det globale ligningssystem, samt evt. efterbehandling.
I kapitlerne 4 og 5 specialiseres det generelle system, d.v.s. Element og Material, til konkrete problemer. Både det lineære potentialelement og det lineært elastiske solidelement formuleres som isoparamtrisk element. Disse to elementer er del af en familie af isoparametriske kontinuumelementer og derfor defineres en superklasse, Continuum, som nedarves fra Element-klassen. En Gausspoint-klasse introduceres til håndtering af de enkelte integrationspunkters koordinater og vægte. Den isoparametriske superklasse specialiseres til 2D og 3D-potentialelementer af serendipitytypen, og i den forbindelse specialiseres Material-klassen til simple lineære materialemodeller. Lineært elastiske solidelementer implementeres af Solid-klassen. Disse elementer nedarves fra potentialelementerne, hvorved modifikationerne hovedsagligt består i redefinering af elementmatricernes dimensioner. Material-klassen specialiseres til lineært, isotropt elastiske materialer i 2D og 3D.
Anden hoveddel af afhandlingen omhandler ikke-lineær elementmetode. I kapitel 6 betragtes løsningen af ikke-lineære ligningssystemer. Først gives generel introduktion ikke-lineær elementmetode i relation til løsningsstrategier. To forskellige typer ikke-lineære problemer identificeres. I den første type problemer kan de interne kræfter kan beregnes eksplicit på basis af et givent estimat på flytningerne. I den anden er det nødvendigt at integrere de konstitutive relationer op over hele belastningsforløbet. Disse problemer er derved karakteriseret ved implicit beregning af de interne kræfter. Iterative strategier for

løsning af ikke-lineære ligninger præsenteres med hovedvægt på buelængdemetoden og orthogonal residual metoden. I forbindelse med udvikling af en applikation, der anvender disse metoder, beskrives supplerende foranstaltninger, der er nødvendige for at sikre en robust kode. Kapitlet afsluttes med at resumere de tilføjelser til Element klassen, som kræves i forbindelse med brugen af de ikke-lineære løsningsstrategier.
I kapitel 7 opstilles elementformuleringen for et geometrisk ikke-lineært stangelement. Elementet baseres på Greens ikke-lineære tøjningsmål og er et eksempel på et problem, hvor de interne kræfter kan beregnes eksplicit. Den ikke-lineære formulering leder til opstilling af en repræsentativ tangentstivhed samt udtryk til beregning af de interne kræfter på baggrund af en given flytningstilstand. Den lineære del af formuleringen giver en elegant formulering af det lineære stangelement. Det lineære stangelement, Bar, anvendes som superklasse for det ikke-lineære element, NlBar. Kapitlet afrundes af 2 eksempler, hvor det totale last-flytningsforløb for ikke-lineære stangkonstruktioner er fastlagt v.h.a. de ikke-lineære algoritmer præsenteret i kapitel 6.
Kapitel 8 omhandler elasto-plastiske materialemodeller som et eksempel på ikke-lineære problemer med implicit beregning af de interne kræfter. Der indledes med en kort introduktion til plasticitetsteori for hærdende materialer samt til integration af de inkrementale konstitutive relationer. Derefter modificeres Element og Gausspoint klasserne, så de kan håndtere problemer, hvor materialeegenskaberne og spændingstilstanden varierer over elementet. En Plastic materialeklasse arver de elastiske relationer fra Elastic, og tilføjer metoder til beregning af de plastiske bidrag til tangentstivheden, samt integration af de konstitutive ligninger i forbindelse med beregning af de interne kræfter. Implementeringen af von Mises plasticitet illustrerer en konkret anvendelse af den elasto-plastiske klasse.
Erfaringerne med objektorienteret programmering i forbindelse med strukturing af elementmetode diskuteres i kapitel 9. Det konkluderes, at objekter kan anvendes til at opdele elementmetodeanalysen i en række uafhængige moduler, som kan udvikles uden, at det påvirker det øvrige system. Selve elementmetodeformuleringen håndteres af FEM-klasserne, som definerer en programmeringskal. Denne programskal muliggør genbrug af store dele af koden, f.eks. input/output, hvorved programmøren er i stand til at koncentrere sig om at implementere selve elementmetodeformuleringen. Systemet muliggør derfor hurtig implementering af nye elementer, materialer eller løsningsalgoritmer. Brugen af de algebraiske klasser og applikationerne stiller ikke særlige krav til programmørens kendskab til objektorienterede principper. Implementering af nye elementer eller materiale modeller kræver derimod, at programmøren er fortrolig med systemet og grundlæggende begreber indenfor objektorienteret programmering. Et kendskab til de basale begreber er derfor forudsætning for at få fuld glæde af de muligheder, som objektorienteret programmering giver i forbindelse med elementmetoden.